Insights from the parallel implementation of efficient algorithms for the fractional calculus

Thesis submitted in accordance with the requirements of the University of Chester for the degree of Doctor of Philosophy

by Nicola Elizabeth Banks

July 2015

CONTENTS

1. Introduction 1
1.1 Content 3

2. Fractional Differential Equations 6
2.1 Introduction and Objectives 6
2.2 Historical interest and application to the 'real world' 6
2.3 Fractional Differential Equations: Definitions 8
2.3.1 Grünwald-Letnikov Definition for the Fractional Derivative 8
2.3.2 Riemann-Liouville Definition for the Fractional Derivative 9
2.3.3 Caputo Definition for the Fractional Derivative 9
2.4 Alternative form for Fractional Differential Equation 10
2.5 Fractional Differential Equations - Examples 11
2.5.1 Example 1 11
2.5.2 Example 2 12
2.6 Laplace Transforms of Fractional Differential Equations 13
2.6.1 Example 1 14
2.6.2 Example 2 14
2.7 Challenges in the solution of fractional differential equations . 15
2.8 Discussion Points 16

3. Numerical Methods for solving differential equations 17
3.1 Introduction and Objectives 17
3.2 What is a numerical method? 17
3.2.1 Numerical Methods: 'Standards' 18
3.3 Numerical Methods for Ordinary Differential Equations 19
3.3.1 Linear Multistep Methods 20
3.3.2 Adams Method 25
3.3.3 Predictor-Corrector Method 32
3.3.4 Runge-Kutta Methods 33
3.4 Numerical Methods for Fractional Differential Equations . . . 35

3.4.1 Lubich's Fractional Multistep Method 37
3.4.2 Fractional Adams Method 44
3.4.3 Diethelm-Chern Algorithm 52
3.5 Challenges with numerical method implementation 62
3.5.1 Improvement of 'accuracy' including Richardson Extrapolation 62
3.6 Discussion Points 64

4. Parallel Computing 65
4.1 Objectives 65
4.2 Introduction to Parallel Computing 65
4.2.1 Flynn's Taxonomy 66
4.3 Why use parallel computers? 69
4.4 Capacity Parallelism 70
4.4.1 Examples of Capacity Parallelism 70
4.5 Implementation of parallel programs 74
4.5.1 Software for parallel programming 75
4.6 Evaluation of Effectiveness 75
4.6.1 Speed up 76
4.6.2 Theorectial versus practical benefit of implementing numerical method in parallel 77
4.6.3 Floating Point Operations (flop) 78
4.7 Challenges in implementing Parallel Computer Programs . . . 79
4.7.1 Scalability 79
4.7.2 Distribution of load 80
4.7.3 Cost 81
4.8 Discussion Points 81

5. MatLab Parallel Computing Toolbox 82
5.1 Objectives 82
5.2 What is 'MatLab Parallel Computing Toolbox' (MPCT)? . . . 82
5.3 Implementation of parallel programs in MPCT 84
5.3.1 Parallel for-loops 85
5.3.2 Batch jobs 87
5.3.3 Distributed Arrays and Single Program Multiple Data 87
5.4 Communication between LABs 89
5.5 Tools to assess efficiency in MPCT 91
5.6 Discussion Points 91

6. Hardware, Software, and Data Collection 92
6.1 Introduction and Objectives 92
6.2 Hardware and Software Specifications 92
6.2.1 Hardware 92
6.2.2 Software 93
6.2.3 Benchmarking Data 93
6.3 Methodology for data collection 93
6.3.1 Efforts to obtain best performance 95
6.4 Opening and Closing the MatLab Pool 95
6.5 Forms of Communication between Workers 96
6.6 Implicit Multithreading 98
6.7 Assessment of Performance 100
6.8 Discussion Points 101

7. Ordinary Differential Equations: Runge-Kutta Method 103
7.1 Introduction and Objectives 103
7.2 Runge-Kutta Method for Ordinary Differential Equations . . . 103
7.2.1 Runge-Kutta Method: The Algorithm 104
7.2.2 Adapted Richardson Extrapolation 105
7.3 Implementation in MatLab 106
7.4 Runge-Kutta Method: Results from MatLab 108
7.4.1 Runge-Kutta Method: Data Tables 113
7.5 Theoretical vs Practical Benefit Calculations 113
7.5.1 FLOP Calculation 116
7.6 Discussion Points 116

8. Diethelm-Chern Algorithm 119
8.1 Objectives 119
8.2 Diethelm-Chern: The Algorithm 119
8.3 Implementation in MatLab 120
8.4 Results in MatLab 123
8.4.1 Diethelm-Chern Algorithm: Data table 124
8.5 Theoretical/Practical Benefits Calculations 124
8.6 FLOPs 128
8.7 Discussion Points 128

9. Fractional Adams Method 129
9.1 Introduction and Objectives 129
9.2 Fractional Adams Method: The Algorithm 129
9.3 Fractional Adams Method: Implementation in MatLab 131
9.4 Results from MatLab 133

9.4.1 Sequential Programs and Multithreading 134
9.4.2 Acceptance of labBroadcast - Idle Workers 137
9.4.3 Fractional Adams Method - Data Table 138
9.5 Theoretical vs Practical Calculations 138
9.5.1 FLOPS 140
9.5.2 Comparison of MPCT Results to C++/MPI 140
9.6 Discussion Points 140

10. Lubich's Fractional Multistep Method 142
10.1 Introduction 142
10.2 Lubich's FMM: Reminder 142
10.2.1 Calculation of Weights 143
10.3 Lubich's Fractional Multistep Method: Implementation in MatLab 143
10.3.1 Results from MatLab 147
10.4 Theoretical vs. Practical Benefit Calculations 153
10.4.1 FLOPS 155
10.5 Alternative MatLab Implementation 156
10.6 Discussion Points 156

11. Conclusion 158
11.1 Key Principles 160
11.2 Ideas for Future Research 160

Appendix 162

A. Runge-Kutta Method for Ordinary Differential Equations - Annotated Program 163

B. Diethelm-Chern - Annotated Program 166

C. Fractional Adams Method (FAM) - Annotated Program 179

D. Lubich's Fractional Multistep Method (FMM) - Annotated Program 185

E. MatLab Programs - CD 202

DECLARATION

No part of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other institution of learning.

ACKNOWLEDGEMENTS

I would like to thank the following people for their support during the course of this research:

- My supervisory team, in particular the guidance provided by Prof Neville Ford and Dr Judy Ford.

- My colleagues and friends in Student Support and Guidance. In particular I would like to acknowledge the support given by Line Manager, Rev Dr Lesley Cooke.

- Last but by no means least my Mum, Brenda A. Banks, for her enouragement and just being there!

ABSTRACT

Insights from the parallel implementation of efficient algorithms for the fractional calculus by Nicola Elizabeth Banks

This thesis concerns the development of parallel algorithms to solve fractional differential equations using a numerical approach. The methodology adopted is to adapt existing numerical schemes and to develop prototype parallel programs using the MatLab Parallel Computing Toolbox (MPCT). The approach is to build on existing insights from parallel implementation of ordinary differential equations methods and to test a range of potential candidates for parallel implementation in the fractional case. As a consequence of the work, new insights on the use of MPCT for prototyping are presented, alongside conclusions and algorithms for the effective implementation of parallel methods for the fractional calculus.

The principal parallel approaches considered in the work include:

- A Runge-Kutta Method for Ordinary Differential Equations including the application of an adapted Richardson Extrapolation Scheme
- An implementation of the Diethelm-Chern Algorithm for Fractional Differential Equations
- A parallel version of the well-established Fractional Adams Method for Fractional Differential Equations
- The adaptation for parallel implementation of Lubich's Fractional Multistep Method for Fractional Differential Equations

An important aspect of the work is an improved understanding of the comparative difficulty of using MPCT for obtaining fair comparisons of parallel implementation. We present details of experimental results which are not satisfactory, and we explain how the problems may be overcome to give meaningful experimental results. Therefore, an important aspect of the conclusions of this work is the advice for other users of MPCT who may be

planning to use the package as a prototyping tool for parallel algorithm development: by understanding how implicit multithreading operates, controls can be put in place to allow like-for-like performance comparisons between sequential and parallel programs.

1. INTRODUCTION

The key aim of this research is the development of prototype parallel programs designed to solve Fractional Differential Equations (FDEs) using a numerical approach. FDEs have been of historical interest to mathematicians but it is only more recently that their uses for modeling processes and systems have been recognised in the wider scientific community. This thesis considers Initial Value Problems (IVPs) of the form:

$$ {}_aD_T^{\alpha} y(t) = f(t, y(t)) \tag{1.1} $$

where:

- α is the order of the fractional derivative,
- t is the independent variable,
- y is a continuous function with initial conditions $y^{(k)}(a) = y_a^{(k)}$ $(k = 0, 1, \ldots, \lceil \alpha \rceil - 1)$,
- $\lceil \alpha \rceil$ is the ceiling function,
- f is a continuous function of two variables,
- the interval for integration is $[a, T]$ where $T > a$.

Numerical methods will be considered over the equispaced grid with points $t_j, j = 0, 1, \ldots, N$. In this grid N is the total number of steps and $t_j = jh$, where h is the step size.

The research in this thesis has been restricted to the scalar case IVP. However, MatLab has been developed for the manipulation of matrices and therefore the prototype programs developed in this research can be adapted for systems of equations.

Initial work described in this thesis considers differential equations of integer order i.e. Ordinary Differential Equations (ODEs). In the ODE case

initial conditions are based on local behaviour. Lessons learned from the ODE case are transfered to differential equations of fractional order i.e. Fractional Differential Equations (FDEs). In the FDE scenario the whole history of the function in weighted form is required in order to solve the equation [7]. This latter property of FDEs makes them applicable to path-dependent models [20] but means analytical solutions are difficult/impossible.

Given the difficulty of solving FDEs analytically, numerical methods for the approximation of FDEs are employed. Low-order numerical methods can produce good results but take time to reach approximate solutions within a desired level of accuracy. High-order methods such as Lubich's Fractional Multistep Method (FMM) are disregarded due to the complexity (and hence the length of time taken) of each step of the algorithm. This research considers a range of numerical methods for approximating FDEs. In particular a key motive of this research is the exploration of how parallel programs can reduce the time required to solve FDEs numerically.

But what is meant by a 'parallel program'? In Chapter 4 we describe what a parallel architecture is and the software needed to operate such technology. In short, a parallel program is a program where all or part of the code can be operated on by multiple 'workers' [1] which are connected together via a network or located in the same device. Later in the thesis we learn more about MatLab Parallel Computing Toolbox (MPCT) as a tool for developing prototype parallel programs (Chapter 5). MPCT was introduced in 2004 and continues to develop its parallel programming abilities. The MatLab programming language is renowned for its operational ease and in particular its ability to solve matrix-based problems. MatLab is used in rapid prototyping with many users requiring resultant code be transformed into FORTRAN or C for static applications [23]. Reasons for needing a static program given in Dubrau et al [23] are: production of a stand-alone executable file using standard/free compiler, integration of the created code into an existing system which does not use MatLab, and/or implementation of the code on a parallel architecture (without the use of MPCT).

One aspect which has come to the fore in this research is the use of implicit multithreading within MPCT. Multithreading is the operation of multiple threads of instruction at one time within the core or processor. So, for example, while you are saving a document (one thread) you can open another document (second thread). The creators of MPCT have built the

[1] A 'worker' is a computing entity such as a computer, or an element of a processor.

multithreading capabilities into the operations of the software without the need for the end-user to 'turn it on'. As a consequence comparison between sequential programs (which will have multithreading in operation) and created parallel programs are not legitimate. The research in this thesis informs the Reader how to resolve the implicit multithreading problem and demonstrates the impact of comparisons to small-scale parallel programs. Where appropriate, evidence from the MPCT User community is provided to understand the results obtained.

The continuing mission of this research has been the further reduction in execution time of the created prototype parallel programs through structural changes and reduction in communication overheads. Each set of programs is assessed for efficiency through the number of 'actions' [2] which are operated within a program and the execution time of the program against its sequential counter-part.

1.1 Content

The following summary describes the content and structure of the remainder of this thesis:

Chapter 2 - Fractional Differential Equations

A general discussion regarding FDEs is provided in **Chapter 2**. The Chapter provides the Riemann-Louville, Caputo, and Grünwald-Letnikov definitions for FDEs. Analytical solutions for simple FDEs are provided with examples. Due to the nature of FDEs, the challenges in solving these equations are discussed providing motivation for the use of numerical methods.

Chapter 3 - Numerical Methods for Differential Equations

Background information regarding numerical methods for ODEs and FDEs are provided, illustrated by examples. The challenges associated with implementing numerical methods are discussed: convergence, consistency, and zero stability. The Richardson Extrapolation Scheme is detailed as a method for improving accuracy which will be later adapted in Chapter 7.

[2] By action we are referring to: addition or subtraction, multiplication or division, fetching or storing data.

Chapter 4 - Parallel Computing

Beginning the background on the computational aspects of this thesis, Chapter 4 provides material regarding parallel computing. A desciption is provided regarding what parallel computing is, including the architecture and software needed to create a parallel environment. Methods for evaluating the effectiveness of resultant parallel programs are discussed along with the challenges of implementing parallel programs. In Section 4.4.1 'multithreading' is defined, an important concept in the later chapters of the thesis.

Chapter 5 - MatLab Parallel Computing Toolbox

Continuing the work of Chapter 4, this Chapter considers the parallel capabilities provided through the MatLab Parallel Computing Toolbox (MPCT) software. Specific constructs for parallel computing are explained, including: parfor, SPMD, codistributed arrays, and batch jobs. The methods of communicating between workers using MPCT are also described. The tools available within MPCT which assist in evaluating performance of created programs are provided.

Chapter 6 - Hardware, Software and Data Collection

This Chapter describes the parallel architecture and software specification used within the research project. The method of data collection with Chapters 7 to 10 are also detailed. The Reader is provided with guidance regarding the impact of multithreading within sequential programs created for comparator purposes. In addition, experimentation of communication techniques between workers within a parallel program (as described in Chapter 5) demonstrate the impact of each command.

Chapter 7 - Ordinary Differential Equations: Runge-Kutta Method

Prototype parallel programs are provided for a Runge-Kutta Method. The accuracy of the resulting Runge-Kutta Method approximations are enhanced through an Adapted Richardson Extrapolation Scheme. An assessment of effectiveness of the prototype parallel programs is undertaken through the calculation of number of actions and comparing the parallel programs to their sequential counter-parts.

Chapter 8 - Diethelm-Chern Algorithm

Chapter 8 develops further the ideas of Chapter 7 by improving load-balance within the prototype parallel programs. The Diethelm-Chern Algorithm is also implemented on an eight-worker architecture which is the first time this is attempted within this research.

Chapter 9 - Fractional Adams Method

The Fractional Adams Method is prototyped in MPCT on both four and eight worker parallel architectures. Experiments are undertaken to improve the execution time of the prototype programs by the acceptance of data on workers which are not active.

Chapter 10 - Lubich's Fractional Multistep Method

Finally Lubich's Fractional Multistep Method (FMM) is implemented in MPCT. Results are provided for hybid prototype programs which operate multithreading during the initial stages of the program. Additionally a set of prototypes are created which restrict multithreading in the initial stages to assess impact upon program performance.

Chapter 11 - Conclusion

Conclusions are drawn regarding MPCT for small-scale parallel architectures. Advice is provided to future MPCT programmers with regard to improvement of performance and viable comparators. Ideas for potential areas of research are also provided.

2. FRACTIONAL DIFFERENTIAL EQUATIONS

2.1 Introduction and Objectives

As mentioned in the introduction to this thesis, Fractional Differential Equations have been of historical interest to mathematicians since the initial correspondence between L'Hopital and Leibniz in 1695 [65]. The more recent interest in FDEs began in the late 1960s and the first book which began the 'new age' was written by Oldham and Spanier in 1974. Oldham and Spanier had been working together since 1968 and had used FDEs to model electrochemical problems [52].

But what is a 'Fractional Differential Equation'? Podlubny [53] describes FDEs as an interpolation of the infinite series of n-fold derivatives and integrals. Baleanu et al [4] explain that, unlike ordinary differential equations (ODEs), FDEs are not local in nature and have a degree of memory.

In more recent decades the use of Fractional Differential Equations (FDEs) for modelling 'real-world' situations has expanded interest to include the wider scientific community. This Chapter aims to provide the Reader background information on FDEs which illustrate why numerical approaches to their solution are desired. To summarise this Chapter satisfies the following objectives:

- To provide examples of the use of FDEs in real-world phenomena
- To provide the standard definitions for FDEs
- To provide examples of analytical solutions for simple FDEs
- To describe the challenges in solving FDEs

2.2 Historical interest and application to the 'real world'

The origins of fractional calculus began with L'Hospital's question "What does $\frac{d^n}{dx^n} f(x)$ mean if $n = \frac{1}{2}$?" in a letter to Leibniz in 1695 [16]. Although

FDEs have remained of interest to mathematicians, it is only more recently that they have become of interest to the wider scientific community. In [65] Weilbeer describes the development of fractional calculus in three stages:

- **Early Stages 1695-1822:** Leibniz correspondence with L'Hospital, Bernoulli, and Wallis. In these letters are references to a derivative "even if its a fraction". Euler develops the Gamma Function and makes reference to fractional derivative. Later there is work by Laplace (fractional derivative of function represented by an integral) and Lacroix (fractional derivatives of power functions). In 1822 Fourier provides a definition of a fractional derivative for a sufficiently "well-behaved" function.

- **1823-1916:** Work using FDEs to solve physical problems began with Abel and the tautochrone problem. "The tautochrone problem consists of the determination of a curve in the (x,y) plane such that the time required for a particle to slide down the curve to its lowest point under gravity is independent of its initial position (x_0,y_0) on the curve" [65]. During this period Liouville developed two definitions for fractional derivatives which led to the Grünwald-Letnikov Definition for the fractional derivative. Riemann's work with the generalisation of the Taylor's series was unified with Liouville's work to provide the Riemann-Liouville definition of fractional differentiation.

- **1916-present:** Marchaud developed an integral version of the Grünwald-Letnikov Definition. Caputo developed his definition of the fractional derivative so that initial conditions which have a physical meaning can be used. Ross organised the first conference on fractional calculus in 1974. Oldham and Spanier [52] published their book "The Fractional Calculus: Theory and Applications of Differentiation and Integration to Arbitrary Order".

As indicated in the list above, mathematicians have had a historical interest in fractional calculus. However it was not until the mid-nineteenth century that fractional calculus was considered with physical problems. In particular the fact that FDEs have memory make these equations useful in modelling 'real-world' situations which are path-dependent [17]. Such situations arise in the physical, biological, and chemical industries. Modelling situations for the wider scientific community reduces the need for experimentation which could be dangerous and heavy on resources.

In the book by Diethelm [16] an extensive list of real-world applications for FDEs is provided. Some of the examples listed are:

- Diffusion processes
- Signal processes
- Modelling of behaviour in viscoelastic or viscoplastic materials under external influences
- Pharmacokinetics
- Modelling of combustion
- Modelling behaviour of human beings

2.3 Fractional Differential Equations: Definitions

As indicated in the historical list above, the fractional derivative does not have a single definition [16]. In this section we will provide three definitions for the fractional derivative for the Reader. To begin with we define the following standard notation which will be used throughout the thesis:

- ${}_aD_T^{\alpha}y\,(t)$ will denote the α derivative of the function $y\,(t)$ where $\alpha \notin \mathbb{N}$. a is the starting terminal.
- ${}_aJ_T^{m-\alpha}y\,(t)$ will denote the α-fold integral of the function $y\,(t)$, conducted over the interval $[a, T]$. Also $m = \lceil \alpha \rceil$

2.3.1 Grünwald-Letnikov Definition for the Fractional Derivative

Dating back to 1867-1868 the Grünwald-Letnikov (GL) Definition can be used in the development of numerical methods [65].

Podlubny [53] and Diethelm [16] provide the GL Definition as follows:

Let $\alpha > 0$, $y\ [a, T] \to \mathbb{R}$ and has $\lceil \alpha \rceil^{th}$ continuous derivative. Additionally $a < t \leq T$. Then,

$$ {}_aD_T^{\alpha}y\,(t) = \lim_{N\to\infty} \Big\{ \frac{N}{T-a} \sum_{k=0}^{N} (-1)^k \binom{\alpha}{k} y\big(t - k[\frac{T-a}{N}]\big) \Big\} \tag{2.1}$$

Podlubny [53] states that manipulation of the GL Definition in its fractional-order backward difference form is not convenient. Podlubny goes on to state the equivalence of the GL Definition to the Riemann-Liouville Definition which follows in Section 2.3.2.

2.3.2 Riemann-Liouville Definition for the Fractional Derivative

The most common definition for fractional derivative is the Riemann-Liouville (RL) Fractional Derivative. Weilbeer [65] states:

Let $\alpha \in \mathbb{R}_+$ and $m = \lceil \alpha \rceil$. The Riemann-Liouville Fractional Differential Operator is,

$$\begin{aligned} {}_aD_T^{\alpha} y(t) &= D_a^m J_T^{m-\alpha} y(t) \\ &= \frac{1}{\Gamma(m-\alpha)} D^m \int_a^T (t-\tau)^{m-\alpha-1} y(\tau)\, d\tau \end{aligned} \tag{2.2}$$

where $a \leq t \leq T$.

To assist in later application of the RL Fractional Differential Operator, the RL Fractional Integral $J_a^{m-\alpha}$ is stated separately for the Reader. In Weilbeer [65]:

Let $\alpha \in \mathbb{R}$ with $a \leq t \leq T$. The Riemann-Liouville Fractional Integral is defined as,

$${}_aJ_T^{m-\alpha} = \frac{1}{\Gamma(m-\alpha)} \int_a^T (t-\tau)^{m-\alpha-1} y(\tau)\, d\tau \tag{2.3}$$

The operator maps $f[a,T] \to \mathbb{R}$ where f is measureable on [a,T] and $\int_a^T |y(\tau)|\, d\tau < \infty$.

The RL Definition for the fractional derivative (Equation 2.2) has been important in the theoretical development of fractional calculus and applications in pure mathematics [53]. However the RL Definition requires knowledge of intial condition information which has no physical meaning. This aspect is particularly important when considering the application of FDEs to real-world situations. As a consequence the Caputo Definition was developed in the late 1960s.

2.3.3 Caputo Definition for the Fractional Derivative

By changing the order of differentiation and integration within the definition of the fractional derivative, Caputo enabled the use of physical measurable

intial condition data such as $y(0)$, $\frac{dy}{dt}$, $\frac{d^2y}{dt^2}$ etc. The Caputo definition is provided as follows [65],[53]:

Let $\alpha \in \mathbb{R}_+$ and $m = \lceil \alpha \rceil$. The Caputo Differential Operator of order α is:

$$\begin{aligned} {}_aD_T^{\alpha} y(t) &= {}_aJ_T^{m-\alpha} D^m y(t) \\ &= \frac{1}{\Gamma(m-\alpha)} \int_a^T (t-\tau)^{m-\alpha-1} D^m y(\tau)\, d\tau \end{aligned} \tag{2.4}$$

where $a \leq t \leq T$.

2.4 Alternative form for Fractional Differential Equation

Beginning with the initial value problem Equation 1.1 and repeated here for the Reader:

$${}_aD_T^{\alpha} y(t) = f(t, y(t))$$

where α is the fractional derivative and initial condition information $y^{(k)}(0) = y_0^{(k)}$.

In the papers by Diethelm [13] & Diethelm and Freed [20] the initial value problem from Equation 1.1 has been interpreted as a Volterra Integral Equation of the second kind with a strongly singular kernel. To do this they alter the order of differentiation and integration in the Rieman-Liouville Definition 2.2 where $m = 1$ to provide the formula:

$${}_aD_T^{\alpha} y(t) = \frac{1}{\Gamma(-\alpha)} \int_a^T \frac{y(\tau)}{(t-\tau)^{\alpha+1}} d\tau \tag{2.5}$$

The Equation 2.5 has a strongly singular kernel which has to be interpreted as a Hadamard finite-part integral. Diethelm and Freed [20] state that numerical methods have only been developed for linear equations in this form. As an alternative they apply a fractional integral operator to produce a Volterra Integral Equation of the second kind with a weakly singular kernal. In other words:

$$y(t) = y_a + \frac{1}{\Gamma(\alpha)} \int_a^T (t-\tau)^{\alpha-1} f(\tau, y(\tau))\, d\tau \tag{2.6}$$

This form of Volterra Integral Equation 2.6 no longer requires the regularisation of the integral. We will see in Chapter 3 that this form of integral equation is valuable in formation of numerical methods.

2.5 Fractional Differential Equations - Examples

To continue our introduction to FDEs let us consider two linear examples which can be solved analytically using the FDE definitions provided. These examples illustrate some of the techniques that are used to solve FDEs.

2.5.1 Example 1

The first example equation to consider is,

$$ {}_0D_t^{\frac{1}{2}} y(t) = -0.5t \tag{2.7} $$

We shall now consider the fractional derivative of Equation 2.7 utilising the Riemann-Liouville definition (Equation 2.2) leads to the following:

$$ {}_0D_t^{\frac{1}{2}} y(t) = D_0^1 J_t^{1-\frac{1}{2}} (-0.5t) $$

Applying the Riemann-Liouville Integral Operator (Equation 2.3) to Equation 2.7,

$$ \begin{aligned} {}_0J_t^{1-\frac{1}{2}} y(t) &= J^{1-\frac{1}{2}} (-0.5t) \\ &= \frac{1}{\Gamma\left(1-\frac{1}{2}\right)} \int_0^t (t-\tau)^{1-\frac{1}{2}-1} (-0.5\tau)\, d\tau \end{aligned} $$

Integrating by parts,

$$
\begin{aligned}
{}_0J_t^{1-\frac{1}{2}} y(t) &= \frac{1}{\Gamma\left(\frac{1}{2}\right)}\left(\left[(-0.5\tau)\frac{(t-\tau)^{\frac{1}{2}}}{\frac{1}{2}}(-1)\right]_0^t - \int_0^t (-1)\frac{(t-\tau)^{\frac{1}{2}}}{\frac{1}{2}}(-0.5)\,d\tau\right) \\
&= \frac{2}{\Gamma\left(\frac{1}{2}\right)}\left([0-0] + \int_0^t (-0.5)(t-\tau)^{\frac{1}{2}}\,d\tau\right) \\
&= \frac{2(-0.5)}{\Gamma\left(\frac{1}{2}\right)}\left[\frac{(t-\tau)^{\frac{3}{2}}}{\frac{3}{2}}(-1)\right]_0^t \\
&= \frac{1}{\Gamma\left(\frac{1}{2}\right)}\left[0 - \frac{2t^{\frac{3}{2}}}{3}\right] \\
&= \frac{-2t^{\frac{3}{2}}}{3\Gamma\left(\frac{1}{2}\right)} \qquad (2.8)
\end{aligned}
$$

Differentiating Equation 2.8 once to complete the solution,

$$
\begin{aligned}
{}_0D_t^{\frac{1}{2}} y(t) &= D^1\left(\frac{-2t^{\frac{3}{2}}}{3\Gamma\left(\frac{1}{2}\right)}\right) \\
&= \frac{3}{2}\frac{-2t^{\frac{1}{2}}}{3\Gamma\left(\frac{1}{2}\right)}
\end{aligned}
$$

$$
y(t) = \frac{-t^{\frac{1}{2}}}{\Gamma\left(\frac{1}{2}\right)} \qquad (2.9)
$$

2.5.2 Example 2

For this second example we shall use the Caputo definition for FDEs (Equation 2.4). The initial problem for differentiating is:

$$
{}_0D_t^{1.3} y(t) = t^2 \qquad (2.10)
$$

Defining the Caputo definition for the fractional derivative,

$$
{}_0D_t^{1.3} y(t) =_0 J_t^{2-1.3} D^2\left(t^2\right)
$$

Undertaking the differentiation element of the right hand side,

$$\begin{aligned} D^2 y(t) &= D^2\left(t^2\right) \\ &= D^1(2t) \\ &= 2 \end{aligned} \tag{2.11}$$

Now performing the integration of Equation 2.11 utilising the RL Fractional Integral Operator (Equation 2.3),

$$\begin{aligned} {}_0D_t^{1.3} y(t) &= {}_0J_t^{0.7}(2) \\ &= \frac{1}{\Gamma(0.7)} \int_0^t (t-\tau)^{0.7-1}(2)\, d\tau \\ &= \frac{2}{\Gamma(0.7)} \left[(-1) \frac{(t-\tau)^{-0.3}}{-0.3} \right]_0^t \end{aligned} \tag{2.12}$$

$$y(t) = \frac{-2t^{-0.3}}{0.3\Gamma(0.7)} \tag{2.13}$$

which completes the solution.

2.6 Laplace Transforms of Fractional Differential Equations

An alternative method for the analytical solution of linear FDEs is obtained through the application of Laplace Transforms. To begin let us provide some definitions of Laplace Transforms for the fractional integral and derivative as provided by Diethelm in [16] where the lower terminal $a = 0$.

It is assumed that $y : [0, \infty) \to \mathbb{R}$ is such that $L[y]$ exists on $[s_0, \infty)$ with $s_0 \in \mathbb{R}$. As previously documented, let $\alpha > 0$ and $m = \lceil \alpha \rceil$. For $s > max\{0, s_0\}$, the Laplace Transformation of the Riemann-Liouville Fractional Integral (equation 2.3) is:

$$L\left[{}_0J_T^{\alpha} y\right](s) = \frac{1}{s^{\alpha}} L[y](s) \tag{2.14}$$

The Laplace Transform of the Caputo Fractional Derivative (Equation 2.4)

$$L\left[{}_0D_T^{\alpha} y\right](s) = s^{\alpha} L[y](s) - \sum_{k=1}^{\lceil \alpha \rceil} s^{\alpha-k} f^{(k-1)}(0) \tag{2.15}$$

2.6.1 Example 1

Let us now consider an example where Laplace Transforms can be used. To begin the FDE under consideration is,

$$ {}_0D_T^{\frac{2}{3}} y(t) = t^2 \tag{2.16}$$

where $y^k(0) = 1$ for $k = 0, 1, 2, ...$ Taking Laplace Transforms of both sides of equation 2.16 gives,

$$ s^{\frac{2}{3}} L[y](s) - \sum_{k=1}^{1} s^{\frac{2}{3}-k} y^{(k-1)}(0) = \frac{2}{s^3} $$

Making $L[y]$ the subject,

$$\begin{aligned} L[y](s) &= \frac{\frac{2}{s^3} + s^{\frac{-1}{3}}}{s^{\frac{2}{3}}} \\ &= \frac{2}{s^3 s^{\frac{2}{3}}} + s^{\frac{-1}{3}} s^{\frac{-2}{3}} \\ &= \frac{2}{s^{\frac{11}{3}}} + s^{-1} \end{aligned}$$

Reversing the Laplace Transform,

$$ y(t) = \frac{2t^{\frac{8}{3}}}{\Gamma\left(\frac{11}{3}\right)} + 1 \tag{2.17}$$

2.6.2 Example 2

Considering a second example where the Laplace Transform can be used. The FDE under consideration is:

$$ {}_0D_T^{\frac{1}{5}} y(t) = 7y(t) \tag{2.18}$$

where $y^k(0) = 1$ for $k = 0, 1, 2, ...$ Taking Laplace Transforms of both sides of equation 2.18 gives,

$$s^{\frac{1}{5}} L[y](s) - \sum_{k=1}^{1} s^{\frac{1}{5}-k} y^{(k-1)}(0) = 7L[y](s)$$

Rearranging the equation,

$$\begin{aligned} s^{\frac{1}{5}} L[y](s) - s^{\frac{-4}{5}} y^{(0)}(0) &= 7L[y](s) \\ s^{\frac{1}{5}} L[y](s) - 7L[y](s) &= s^{\frac{-4}{5}} \end{aligned}$$

Now making $L[y]$ the subject,

$$\begin{aligned} L[y]\left(s^{\frac{1}{5}} - 7\right) &= s^{\frac{-4}{5}} \\ L[y] &= \frac{s^{\frac{-4}{5}}}{s^{\frac{1}{5}} - 7} \end{aligned}$$

Using the Laplace Transform for the one-parameter Mittag-Leffler function ($E_\alpha(-\beta t^\alpha)$ see [16] and [53]) which is,

$$L[y](s) = \frac{s^{\alpha-1}}{s^\alpha + \beta} \tag{2.19}$$

The reverse Laplace Transform of our example is,

$$y(t) = E_{\frac{1}{5}}\left(7t^{\frac{1}{5}}\right) \tag{2.20}$$

2.7 Challenges in the solution of fractional differential equations

The analytical solutions provided earlier demonstrated that simple FDEs can be solved using analytical means. The Laplace Transform Method is only applicable to linear differential equations. So what about the non-linear case? What would happen if the function y was not integrable in the fractional derivative definitions [53]? We are already aware that the initial condition information needed to use the Riemann-Liouville definition does not have a

physical meaning and so the Caputo approach is preferred for 'real-world' applications. With these questions in mind we look to numerical method for solving differential equations in Chapter 3.

One final note: In Diethelm's paper [14], the Author emphasises that initial condition information is not just in the locality but the entire history of the problem which makes solution complex. This complexity gives extra motivation for solving FDEs using some kind of computer architecture. The speed of solution of any numerical method may be hindered given complexity of initial condition data which may make a parallel architecture more desireable. Again this will be explored further in Chapter 8 to Chapter 10.

2.8 Discussion Points

This Chapter has provided a brief introduction to the Fractional Derivative. We have looked at definitions for the fractional derivative and briefly considered analytical solutions. Overall this Chapter has served as motivation to using a numerical approach to solving FDEs. To summarise, the following points will be taken forward in this thesis:

- The Caputo definition for the fractional derivative which utilises initial condition information which is physically measureable.
- The interpretation of FDEs as Volterra Integral Equations of the second kind which can be used to form numerical methods for FDEs
- The challenges in analytically solving FDEs

3. NUMERICAL METHODS FOR SOLVING DIFFERENTIAL EQUATIONS

3.1 Introduction and Objectives

Returning to Equation 1.1 in Chapter 1 and repeated here for the Reader:

$$ {}_aD_T^{\alpha} y(t) = f(t, y(t)) $$

where $y^{(k)}(a) = y_a^{(k)}$ and $k = 0, 1, ..., \lceil \alpha \rceil$. In the case where α is of integer order, Equation 1.1 becomes an ordinary differential equation which provides a useful starting point for numerical methods for later adaptation to the fractional case. Therefore this Chapter aims to explore a selection of numerical methods for differential equations which will be later translated into parallel programs. In addition this Chapter will provide the Reader with general information regarding numerical methods, challenges faced when implementing these methods, and a way to improve accuracy i.e. Richardson Extrapolation. In summary, the objectives of this Chapter are:

- To outline a selection of numerical methods for ODEs
- To outline a selection of numerical methods for FDEs
- To provide the terminology used within the implementation of numerical methods
- To present the challenges faced when implementing numerical methods
- To describe how improvements can be made in the accuracy of results obtained through numerical methods using the Richardson Extrapolation Scheme

3.2 What is a numerical method?

In Chapter 2 examples of where Fractional Differential Equations (FDEs) have been used to model 'real-world' situations found in the physical, biological and chemical industries were provided. Ordinary differential equations

(ODEs) have been used for modelling 'real-world' situations for a longer period of time and have been studied widely. In both FDE and ODE cases, analytical or approximate methods can be difficult or complex to solve and consequently numerical methods are sought.

But what is a 'numerical method'? A numerical method is a numerical procedure used to approximate the solution to a problem with given initial or boundary value conditions. In the context of this thesis a numerical method takes the form of a difference equation used to solve time dependent differential equations of integer or fractional order. The problem is solved under a continuous interval $[a, T]$ which is discretised into steps of size h. Lambert [40] describes the difference equation as containing consecutive approximations y_{n+j} where $j = 0, 1, ..., k$ which are required to calculate the next element of the sequence until the final destination (T) is reached. Where $k > 1$ the method is described as a multistep method (see Section 3.3.1) and where $k = 1$ the method is described as a one-step method e.g. Forward Euler Method.

In this thesis the size of h remained constant within the 'current' incarnation of the method. Other researchers have considered varied step sizes in a bid to control error or reduce computational cost (see for example [25]).

3.2.1 Numerical Methods: 'Standards'

Consistency, convergence and stability are standards required for a numerical method to achieve a desired level of accuracy to the exact solution. In order to understand these standards let us consider Equation 1.1 in the form:

$$Dy(t) - f(t, y(t)) = 0 \tag{3.1}$$

where the exact solution is $y(t)$. Now suppose there is a numerical solution with step size h where $t = jh$ with $j = 0, 1, ..., k$ (k is the terminal step). This numerical solution is called y_{n+j} and approximates the exact solution.

Consistency

Consistency means the approximate solution 'nearly' satisifies the exact solution i.e.,

$$Dy_{n+j} - f\left(jh, y_{n+j}\right) = R_{n+j},$$

where R_{n+j} is the residual which tends to zero as $h \to 0$.

Convergence

Convergence means the approximate solution y_{kh} tends to the exact solution $y\left(t\right)$ as $h \to 0$. A necessary and sufficient condition for convergence is the numerical method should be both consistent and zero-stable [40]. Stability is defined in the section below.

Stability

Lambert [40] provides definitions for stability. Essentially stability is concerned with the sensitivity of the numerical method to small perturbations to the initial value problem. If a numerical method is not stable it will amplify the perturbation in the resultant approximation therefore rendering the solution inaccurate. Perturbations can occur during discretisation or when terms are calculated (round-off errors). In particular, a numerical method is termed *zero-stable* if for all $t \in [a, T]$ there exists S which is a real constant satisfying the inequalities 3.2 and 3.3. In these inequalities δ_n and δ_n^* are two perturbations to the numerical method with corresponding solutions z_n and z_n^*. Additionally $k = 0, 1, ..., n$ where n is the terminal step, and the step size $h \in (0, h_0]$.

$$\|z_n - z_n^*\| \leq S\epsilon \tag{3.2}$$

whenever

$$\|\delta_n - \delta_n^*\| \leq \epsilon. \tag{3.3}$$

3.3 Numerical Methods for Ordinary Differential Equations

As discussed earlier, considering the case where α is of integer order (Equation 1.1) provides a useful starting point for exploration of the numerical methods for differential equations. The examples in this Section will be concerned with the case where $\alpha = 1$, i.e.:

$$y^{'}\left(t\right) = f\left(t, y\left(t\right)\right), \text{ where } y\left(0\right) = y_0. \tag{3.4}$$

The subsequent information provides a general summary of numerical methods used within this project. The Linear Multistep Method and the Adams Method are later adapted for FDEs in Section 3.4. In Chapter 7 the Runge-Kutta Method is implemented in a parallel computing architecture.

3.3.1 Linear Multistep Methods

The standard definition for the Linear Multistep Method (LMM) is:

$$\sum_{j=0}^{k} \alpha_j y_{n+j} = h \sum_{j=0}^{k} \beta_j f_{n+j}, \tag{3.5}$$

where $f_{n+j} = f\left(t_{n+j}, y_{n+j}\right)$, $y_{n+j} = y\left(t_{n+j}\right)$ and $j = 0, 1, \ldots, k$. The coefficients or *'weights'* α_j and β_j are constants subject to the following conditions:

$$\alpha_k = 1 \tag{3.6}$$

$$|\alpha_0| + |\beta_0| \neq 0. \tag{3.7}$$

The condition 3.6 prevents the LMM being altered by multiplication through the equation, and condition 3.7 ensures a genuine k step method [57]. If $\beta_k = 0$ the LMM is explicit i.e. the solution which is being sought does not also appear on the functional side of the method. From reference [31] an explicit 2-step LMM example is given, Equation 3.8 below. In this example two pieces of data will be required to start the method.

$$y_{n+2} - y_{n+1} = \frac{h}{2}\left(3f\left(t_{n+1}, y_{n+1}\right) + f\left(t_n, y_n\right)\right) \tag{3.8}$$

A numerical method is implicit if $\beta_k \neq 0$ i.e. the method is non-linear. Lambert [40] provides an example of an implicit two-step Method which requires two pieces of data to instigate the scheme.

$$y_{n+2} + y_{n+1} - 2y_n = \frac{h}{4}\left(f\left(t_{n+2}, y_{n+2}\right) + 8f\left(t_{n+1}, y_{n+1}\right) + 3f\left(t_n, y_n\right)\right) \tag{3.9}$$

LMM: Derivation

The LMM definition is derived from the interpolation of previously calculated solutions and functional values. Ralston et al [55] provide a definition for the numerical quadrature when approximating the definite integral which can be manipulated to provide the LMM definition. Using $Y(t)$ to denote the exact solution and t_{ij} being intermediate steps, the definition for approximating the definite integral is,

$$y_n = \sum_{j=0}^{k} \sum_{p=0}^{m_j} A_{jp} y_{n-j}^{(p)} \tag{3.10}$$

where A_{jp} are coefficients yet to be defined. Considering the case where $m_j = 1$:

$$\begin{aligned} y_n &= \sum_{j=0}^{k} \sum_{p=0}^{1} A_{jp} y_{n-j}^{(p)} \\ &= \sum_{j=0}^{k} \left(A_{j0} y_{n-j}^{(0)} + A_{j1} y_{n-j}^{'} \right) \\ &= \sum_{j=0}^{k} A_{j0} y_{n-j}^{(0)} + \sum_{j=0}^{k} A_{j1} y_{n-j}^{'} \end{aligned}$$

Rearranging the equation above yields,

$$\sum_{j=0}^{k} \alpha_j y_{n-j} = h \sum_{j=0}^{k} \beta_j y_{n-j}^{'}$$

where:

- h is the step size defined as $h = t_{n+j} - t_{n+j-1}$
- $\sum_{j=0}^{k} \alpha_j = (1 - A_{00}) - \sum_{j=1}^{k} A_{j0}$
- $h \sum_{j=0}^{k} \beta_j = \sum j = 0^k A_{j1}$

LMM: First and Second Characteristic Polynomials

The equation 3.5 can be rewritten using *First and Second Characteristic Polynomials* [9]. The Characteristic Polynomials are defined as:

First Characteristic Polynomial:

$$\rho\left(\frac{1}{z}\right) = \sum_{j=0}^{k} \alpha_j \left(\frac{1}{z}\right)^j = \alpha_0 \left(\frac{1}{z}\right)^0 + \alpha_1 \left(\frac{1}{z}\right)^1 + \alpha_2 \left(\frac{1}{z}\right)^2 + \dots \tag{3.11}$$

Second Characteristic Polynomial:

$$\sigma\left(\frac{1}{z}\right) = \sum_{j=0}^{k} \beta_j \left(\frac{1}{z}\right)^j = \beta_0 \left(\frac{1}{z}\right)^0 + \beta_1 \left(\frac{1}{z}\right)^1 + \beta_2 \left(\frac{1}{z}\right)^2 + \dots \tag{3.12}$$

where k is the order and z is the Backward Difference Operator.

Considering Example 3.8 the First Characteristic Polynomial would be,

$$\begin{aligned} \rho\left(\frac{1}{z}\right) &= \left(0\right)\left(\frac{1}{z}\right)^0 + \left(-1\right)\left(\frac{1}{z}\right)^1 + \left(1\right)\left(\frac{1}{z}\right)^2 \\ &= \frac{-1}{z} + \frac{1}{z^2} \\ &= \frac{1-z}{z^2} \end{aligned} \tag{3.13}$$

The Second Characteristic Polynomial for Example 3.8 would be,

$$\begin{aligned} \sigma\left(\frac{1}{z}\right) &= \left(-1\right)\left(\frac{1}{z}\right)^0 + \left(\frac{3}{2}\right)\left(\frac{1}{z}\right)^1 + \left(0\right)\left(\frac{1}{z}\right)^2 \\ &= \left(-1\right) + \left(\frac{3}{2z}\right) \\ &= \frac{3-2z}{2z} \end{aligned} \tag{3.14}$$

Conditions for Consistency

Following on from the definition of Consistency given in Section 3.2.1 we now provide conditions for consistency given in [40] and [31]. These conditions are:

$$\sum_{j=0}^{k} \alpha_j = 0 \tag{3.15}$$

$$\sum_{j=0}^{k} j\alpha_j - \beta_j = 0 \tag{3.16}$$

Alternative definition for Zero-Stability

An alternative definition for Zero-Stability is given in the literature ([40], [30], and [9]). This alternative defintion requires that a numerical method should satisfy the *root condition*. By *root condition* we are referring to all roots of the First Characteristic Polynomial (Equation 3.11) having modulus less than or equal to unity and that those which are equal to unity should be simple (i.e. not repeated).

Forward Shift Notation and Characteristic Polynomial

Using Forward Shift Operator notation defined in Lambert [40],

$$\begin{aligned} Ef_n &= f_{n+1} \\ E^2 f_n = E(Ef_n) &= Ef_{n+1} = f_{n+2} \end{aligned} \tag{3.17}$$

In addition Lambert defines π, a polynomial of degree k which can be written as,

$$\pi(r) = \sum_{j=0}^{k} \gamma_j r^j \tag{3.18}$$

Then extending the forward shift operator notation yields:

$$\pi(E) f_n = \sum_{j=0}^{k} \gamma_j f_{n+j} \tag{3.19}$$

Following on from Equation 3.19, the LMM can be rewritten in terms of the First and Second Characteristic Polynomials as,

$$\omega(E)\, y_n = h\rho(E)\, f_n. \tag{3.20}$$

The First and Second Characteristic polynomials are used to form the function $\omega(z)$ which is defined as:

$$\omega(z) = \frac{\sigma\left(\frac{1}{z}\right)}{\rho\left(\frac{1}{z}\right)}. \tag{3.21}$$

The Taylor's expansion of $\omega(z)$ provide the convolution weights for a quadrature formula. Returning to our Example 3.8 and our calculated First and Second characteristic polynomials (Equations 3.13 and 3.14), the $\omega(z)$ formula would be,

$$\begin{aligned}
\omega(z) &= \frac{\left(\frac{3-2z}{2z}\right)}{\left(\frac{1-z}{z^2}\right)} \\
&= \left(\frac{3-2z}{2z}\right)\left(\frac{z^2}{1-z}\right) \\
&= \frac{z}{2}\frac{(3-2z)}{(1-z)} \\
&= \frac{z}{2}(3-2z)(1-z)^{-1}
\end{aligned}$$

Now conducting a binomial expansion of the final set of brackets gives,

$$\begin{aligned}
\omega(z) &= \frac{z}{2}(3-2z)\left(1+(-1)(-z)+\frac{(-1)(-1-1)}{2!}(-z)^2+\frac{(-1)(-1-1)(-1-2)}{3!}(-z)^3+\ldots\right) \\
&= \frac{z}{2}(3-2z)\left(1+z+z^2+z^3+\ldots\right)
\end{aligned}$$

Expanding the brackets,

$$\begin{aligned}\omega(z) &= \frac{z}{2}\left(3-2z+3z-2z^2+3z^2-2z^3+3z^3-\ldots\right)\\ &= \frac{z}{2}\left(3+z+z^2+z^3+\ldots\right)\\ &= \frac{3z}{2}+\frac{z^2}{2}+\frac{z^3}{2}+\ldots \end{aligned} \tag{3.22}$$

From Equation 3.22 the convolution weights are provided as:

$$\begin{aligned}\omega_0 &= 0\\ \omega_1 &= \frac{3}{2}\\ \omega_2 &= \frac{1}{2}\\ \omega_3 &= \frac{1}{2}\\ &\vdots\end{aligned}$$

In the paper [43], Lubich states that convolution quadratures have "excellent" stability properties. Lubich also indicates the following areas in which convolution quadratures work well:

1. when there is a singular kernel.
2. where the kernel has multiple time scales.
3. where the kernel is highly oscillatory.

3.3.2 Adams Method

In [3] a definition for the Adams Method is provided derived from integrating the initial value problem (Equation 3.4) over the continuous interval $[t_n, t_{n+1}]$. The resultant equation is provided below:

$$\int_{t_n}^{t_{n+1}} y'(\tau)\,d\tau = \int_{t_n}^{t_{n+1}} f(\tau, y(\tau))\,d\tau. \tag{3.23}$$

Integrating the left hand side,

$$\begin{aligned}
[y(\tau)]_{t_n}^{t_{n+1}} &= \int_{t_n}^{t_{n+1}} f(\tau, y(\tau))\, d\tau \\
y(t_{n+1}) - y(t_n) &= \int_{t_n}^{t_{n+1}} f(\tau, y(\tau))\, d\tau \\
y(t_{n+1}) &= y(t_n) + \int_{t_n}^{t_{n+1}} f(\tau, y(\tau))\, d\tau. \qquad (3.24)
\end{aligned}$$

Following work earlier in this Chapter, the integral in Equation 3.24 is now replaced by quadrature. In other words the integral is placed by interpolating polynomials containing the previously computed values of $f(t_{n+j}, y_{n+j})$. Using shortened notation $y_{n+j} = y(t_{n+j})$ Equation 3.24 becomes:

$$y_{n+1} = y_n + h \sum_{j=0}^{k} \beta_j f_{n+j}. \qquad (3.25)$$

The equation 3.25 is the general Adams Method formula. It is visible that the Adams Method is a form of LMM where $\alpha_0 = -1$, $\alpha_1 = 1$, and $\alpha_2, ..., \alpha_k = 0$. The values of β_j determine the number of steps in the Method and therefore the order. The higher the order the quicker the convergence.

The Explicit Adams Methods are called Adams-Bashforth Methods, the most common example being the Forward Euler Method $y_{n+1} - y_n = hf_n$. The First Characteristic Polynomial $\rho\left(\frac{1}{z}\right)$ for the Forward Euler Method is,

$$\begin{aligned}
\rho\left(\frac{1}{z}\right) &= (-1)\left(\frac{1}{z}\right)^0 + (1)\left(\frac{1}{z}\right)^1 \\
&= (-1) + \frac{1}{z} \\
&= \frac{1-z}{z} \qquad (3.26)
\end{aligned}$$

The Second Characteristic Polynomial $\omega\left(\frac{1}{z}\right)$ for the Forward Euler Method is,

$$\begin{aligned}
\sigma\left(\frac{1}{z}\right) &= (1)\left(\frac{1}{z}\right)^0 + (0)\left(\frac{1}{z}\right)^1 \\
&= 1 \qquad (3.27)
\end{aligned}$$

As before the convolution weights can now be constructed through the formation of function $\omega(z)$:

$$\begin{aligned}\omega(z) &= \frac{1}{\frac{1-z}{z}} \\ &= \frac{z}{1-z} \\ &= z(1-z)^{-1} \end{aligned} \tag{3.28}$$

Using the binomial expansion on the final set of brackets,

$$\begin{aligned}\omega(z) &= z\left(1+(-1)(-z)+\frac{(-1)(-1-1)}{2!}(-z)^2+\frac{(-1)(-1-1)(-1-2)}{3!}(-z)^3+\ldots\right) \\ &= z(1+z+z^2+z^3+\ldots) \\ &= z+z^2+z^3+z^4+\ldots \end{aligned} \tag{3.29}$$

From Equation 3.29 we can now extract the convolution weights below,

$$\begin{aligned}\omega_0 &= 0 \\ \omega_1 &= 1 \\ \omega_2 &= 1 \\ \omega_3 &= 1 \\ \omega_4 &= 1\end{aligned}$$

Consistency and Convergence of the Forward Euler Method

Using Equations 3.15 and 3.16 the consistency of the Forward Euler Method can be established. Using Equation 3.15 we obtain the following:

$$\begin{aligned}\sum_{j=0}^{1}\alpha_j &= \alpha_0+\alpha_1 \\ &= (-1)+1=0\end{aligned}$$

Thus satisfying the first condition of consistency. Now using Equation 3.16,

$$\begin{aligned}\sum_{j=0}^{1} j\alpha_j-\beta_j &= (0\alpha_0-\beta_0)+(1\alpha_1-\beta_1) \\ &= (0-1)+(1-0) \\ &= -1+1=0\end{aligned}$$

Thus satisfying the second condition of consistency. Therefore the Forward Euler Method is consistent. As all roots to the First Characteristic polynomial have $\|1\|$ and are simple, the Forward Euler Method satisfies the root condition and is zero stable. Therefore as the Forward Euler Method is consistent and zero stable, the Method is convergent.

The Implicit Adams Methods are called Adams-Moulton Methods, the most common examples are the Backward Euler Method $y_{n+1} = y_n + hf_{n+1}$ and the Trapezium Rule $y_{n+1} - y_n = \frac{h}{2}(f_{n+1} + f_n)$. Again let us construct the First and Second Characteristic Polynomials for both the Backward Euler Method and Trapezium Rule.

Backward Euler Method

The First Characteristic Polynomial $\rho\left(\frac{1}{z}\right)$ for the Backward Euler Method is:

$$\begin{aligned}
\rho\left(\frac{1}{z}\right) &= \sum_{j=0}^{1} \alpha_j \left(\frac{1}{z}\right)^j \\
&= \alpha_0 \left(\frac{1}{z}\right)^0 + \alpha_1 \left(\frac{1}{z}\right)^1 \\
&= (-1) + (1)\left(\frac{1}{z}\right) \\
&= \frac{1-z}{z}. \qquad (3.30)
\end{aligned}$$

The Second Characteristic Polynomial $\sigma\left(\frac{1}{z}\right)$ for the Backward Euler Method is:

$$\begin{aligned}
\sigma\left(\frac{1}{z}\right) &= \sum_{j=0}^{1} \beta_j \left(\frac{1}{z}\right)^j \\
&= \beta_0 \left(\frac{1}{z}\right)^0 + \beta_1 \left(\frac{1}{z}\right)^1 \\
&= (0).(1) + (1).\left(\frac{1}{z}\right) \\
&= \frac{1}{z}. \qquad (3.31)
\end{aligned}$$

Using equations 3.30 and 3.31 to provide the convolution weights for the example:

$$\begin{aligned}\omega(z) &= \left(\frac{\frac{1}{z}}{\left(\frac{1-z}{z}\right)}\right) \\ &= \frac{1}{z}.\frac{z}{1-z} \\ &= (1-z)^{-1} \end{aligned} \tag{3.32}$$

Using the binomial expansion Equation 3.32 becomes,

$$\begin{aligned}\omega(z) &= \left(1+(-1)(-z)+\frac{(-1)(-1-1)}{2!}(-z)^2+\frac{(-1)(-1-1)(-1-2)}{3!}(-z)^3+\ldots\right) \\ &= 1+z+z^2+z^3+\ldots \end{aligned} \tag{3.33}$$

Equation 3.33 now leads to the following convolution weights,

$$\begin{aligned}\omega_0 &= 1 \\ \omega_1 &= 1 \\ \omega_2 &= 1 \\ \omega_3 &= 1\end{aligned}$$

Consistency and Convergence of Backward Euler Method

Using Equations 3.15 and 3.16 the Backward Euler Method can be tested for consistency. Beginning with Equation 3.15 we observe that, as with all Adam's Methods, the values for α_0 and α_1 remain the same as that of the Forward Euler Method and therefore this condition is satisfied.

Returning to Equation 3.16 we can now test the Backward Euler Method for the second condition for consistency.

$$\begin{aligned}\sum_{j=0}^{1} j\alpha_j - \beta_j &= (0\alpha_0-\beta_0)+(1\alpha_1-\beta_1) \\ &= (0-0)+(1-1)=0\end{aligned}$$

Thus the Backward Euler Method is consistent. The roots of the First Characteristic Polynomial are $\|1\|$ and are simple, the Method is therefore zero stable. Consequently we know the Backward Euler Method satisfies the conditions for convergence.

Trapezium Rule

The First Characteristic Polynomial for the Trapezium Rule is:

$$\begin{aligned}\rho\left(\frac{1}{z}\right) &= (-1)\left(\frac{1}{z}\right)^{0} + (1)\left(\frac{1}{z}\right)^{1} \\ &= (-1) + \frac{1}{z} \\ &= \frac{1-z}{z}\end{aligned} \tag{3.34}$$

The Second Characteristic Polynomial for the Trapezium Rule is:

$$\begin{aligned}\sigma\left(\frac{1}{z}\right) &= \left(\frac{1}{2}\right)\left(\frac{1}{z}\right)^{0} + \left(\frac{1}{2}\right)\left(\frac{1}{z}\right)^{1} \\ &= \frac{1}{2} + \frac{1}{2z} \\ &= \frac{1}{2}\left(\frac{1+z}{1-z}\right) \\ &= \frac{1}{2}(1+z)(1-z)^{-1}\end{aligned} \tag{3.35}$$

Using Equations 3.34 and 3.35 provides the function $\omega(z)$ which is,

$$\begin{aligned}\omega\left(z\right) &= \frac{\frac{z+1}{2z}}{\frac{1-z}{z}} \\ &= \left(\frac{z+1}{2z}\right)\left(\frac{z}{1-z}\right) \\ &= \frac{1}{2}\left(\frac{z+1}{1-z}\right)\end{aligned} \tag{3.36}$$

Conducting the binomial expansion of the final set of brackets gives,

$$\begin{aligned}
\omega\Big(z\Big) &= \frac{1}{2}(1+z)\left(1+(-1)(-z)+\frac{(-1)(-1-1)}{2!}(-z)^2+\frac{(-1)(-1-1)(-1-2)}{3!}(-z)^3+\ldots\right) \\
&= \frac{1}{2}(1+z)\left(1+z+z^2+z^3+\ldots\right) \\
&= \frac{1}{2}\left(1+z+z+z^2+z^2+z^3+z^3+z^4+\ldots\right) \\
&= \frac{1}{2}\left(1+2z+2z^2+2z^3+\ldots\right) \\
&= \frac{1}{2}+z+z^2+z^3+\ldots \qquad (3.37)
\end{aligned}$$

Equation 3.37 now leads to the convolution weights which are,

$$\begin{aligned}
\omega_0 &= \frac{1}{2} \\
\omega_1 &= 1 \\
\omega_2 &= 1 \\
\omega_3 &= 1
\end{aligned}$$

Consistency and Convergence of the Trapezium Rule

Using Equations 3.15 and 3.16 we are able to determine the consistency of the Trapezium Rule. We note, as with all Adams Methods, that the first condition of consistency (Equation 3.15) is satisfied.

Returning to Equation 3.16 and applying to the Trapezium Rule,

$$\begin{aligned}
\sum_{j=0}^{1} j\alpha_j-\beta_j &= (0\alpha_0-\beta_0)+(1\alpha_1-\beta_1) \\
&= \left(0-\frac{1}{2}\right)+\left(1-\frac{1}{2}\right) \\
&= -\frac{1}{2}+1-\frac{1}{2}=0
\end{aligned}$$

Thus the Trapezium Rule satisfies the second condition of consistency. Therefore the Trapezium Rule is consistent. We also note the roots of the First Characteristic Polynomial are $\|1\|$ and are simple. Consequently the Trapezium Rule is zero stable. As the Trapezium Rule is both consistent and zero stable we accept the Method is also convergent.

3.3.3 Predictor-Corrector Method

Implicit LMMs require the solution of a non-linear equation i.e. the approximate solution appears as the solution which is being calculated and in the function used in the calculation. For example consider the implicit Trapezium Rule provided in the previous section:

$$y_{n+1} - y_n = \frac{h}{2}\left(f_{n+1} + f_n\right), \tag{3.38}$$

where $f_{n+1} \approx f\left(t_{n+1}, y_{n+1}\right)$. Therefore equation 3.38 has y_{n+1}, the approximation which we wish to obtain, appearing as the result (left side of 3.38) and as part of the function f on the right side. Therefore how will y_{n+1} be evaluated?

One way to resolve the problem of a nonlinear equation is to use an explicit LMM to provide an approximation of the non-linear element before proceeding. This initial explicit LMM is called a *'predictor'* and the subsequent implicit LMM is termed the *'corrector'*. The coupling of LMMs in this form are termed *Predictor-Corrector Methods*.

In [40] a general formula is provided for the Predictor-Corrector Methods:

$$\sum_{j=0}^{k} \alpha_j^* y_{n+j} = h \sum_{j=0}^{k-1} \beta_j^* f_{n+j} \tag{3.39}$$

$$\sum_{j=0}^{k} \alpha_j y_{n+j} = h \sum_{j=0}^{k-1} \beta_j f_{n+j} + h\beta_k f_{n+k}^*, \tag{3.40}$$

where $f_{n+K}^* = f\left(t_{n+k}, y_{n+k}^P\right)$ obtained through the application of the 'predictor' Equation 3.39. The weights of the predictor are indicated using the '*' superscript. Equation 3.40 is the corrector element of the pairing.

Although the Predictor-Corrector Method above indicates a single application of the pairing, the corrector can be applied successively. For example:

- "Correcting to convergence" - The predictor equation is initially applied, the corrector is repeatedly applied until convergence to some predefined error is satisfied. This form of Predictor-Corrector Pair is subject to potentially long periods of computation dependent on the convergence properties of the underlying LMMs.

- $P\,(EC)^{\mu}\,E^{1-r}$ - The application of the predictor (P) initially is followed by successive evaluation of the function (E) and corrector method (C). NB. μ is a positive integer referring to the number of times the EC part of the pairing is applied. The final 'E' refers to a final evaluation of the function which may or may not be needed ($r = 0$ or $r = 1$) [40].

A Predictor-Corrector pairing is common between the Adams-Bashford Method and the Adams-Moulton Method. This form of Predictor-Corrector Method is considered for Fractional Differential Equations later in this Chapter and thesis (see Chapter 9).

An example Predictor-Corrector pairing is provided in Lambert [40] and is given below:

Corrector

$$y_{n+2} - y_n = h\bigg(f\left(t_{n+2}, y^*_{n+2}\right) + f\left(t_n, y_n\right)\bigg) \tag{3.41}$$

where the **Predictor** for y^*_{n+2} is given by,

$$y^*_{n+2} - 3y_{n+1} + 2y_n = \frac{h}{2}\bigg(f\left(t_{n+1}, y_{n+1}\right) - 3f\left(t_n, y_n\right)\bigg). \tag{3.42}$$

3.3.4 Runge-Kutta Methods

In Burrage [10] Runge-Kutta Methods (shortened to RK Methods) are described as a family of multistage one-step methods. By 'multistage' we are referring to the calculation of one step (e.g. y_n to y_{n+1}) using additional 'off-step' functional approximations. This approach differs from Linear Multistep Methods where previously evaluated approximations (y_1, y_2,..., y_n, and potentially y_{n+1} itself) are used to evaluate the current approximation y_{n+1}. The number of 'off-step' functional approximations s determines the accuracy of the RK Method. As the values of s increases so does the order of convergence. In [40] the general s-stage Runge-Kutta structure is provided as:

$$y_{j+1} = y_j + h\sum_{i=1}^{s} b_i K_i, \tag{3.43}$$

where $i = 1, 2, ...s$ and:

$$K_i = f\left(t_j + hc_i, y_j + h\sum_{n=1}^{s} a_{i,n}K_n\right). \tag{3.44}$$

As can be seen in 3.44 K_i are the 'off-step' approximations of solutions at $t_j + hc_1, \ldots, t_j + hc_s$. The coefficients of equations 3.43 and 3.44 can be displayed using the Butcher Array as:

c_1	$a_{1,1}$	$a_{1,2}$	...	$a_{1,s}$
c_2	$a_{2,1}$	$a_{2,2}$	...	$a_{2,s}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
c_s	$a_{s,1}$	$a_{s,2}$	...	$a_{s,s}$
	b_1	b_2	...	b_s

This can also be expressed as:

C	A
	B

The form above makes it clear that components of C are row sums of A. In other words:

$$c_i = \sum_{n=1}^{s} a_{i,n}$$

From [40] please note the following:

- If the matrix A is strictly lower triangular i.e. $a_{i,n} = 0$ for $n \geq i$ the RK Method is explicit.
- If the matrix A is not lower triangular i.e. $a_{i,n} \neq 0$ for $n > i$ the RK Method is implicit. In this scenario an implicit system of dimension ms has to be solved.
- A compromise is the semi-implicit case where matrix A is lower triangular i.e. $a_{i,n} = 0$ for $n > i$. This scenario would result in s uncoupled system of dimension m.

An example Runge-Kutta Method is given in Lambert [40] as:

$$y_{n+1} - y_n = \frac{h}{4}\left(K_1 + 3K_3\right) \tag{3.45}$$

n	K_1	K_2	K_3	y_{n+1}
0	0.21	0.707	0.204	0.721
1	0.201	0.198	0.195	0.741
2	0.192	0.189	0.185	0.760
3	0.182	0.179	0.176	0.778
4	0.173	0.169	0.166	0.795

Tab. 3.1: Example Runge-Kutta Method Results

where

$$K_1 = f(t_n, y_n) \tag{3.46}$$

$$K_2 = f\left(t_n + \frac{h}{3}, y_n + \frac{hK_1}{3}\right) \tag{3.47}$$

$$K_3 = f\left(t_n + \frac{2h}{3}, y_n + \frac{2hK_2}{3}\right) \tag{3.48}$$

Considering the initial value problem,

$$D^{'} y(t) = y(1 - y)$$

where $y(0) = y_0 = 0.7$. For step size $h = 0.1$ the first five steps in the above RK Method are summarised in Table 3.1.

3.4 Numerical Methods for Fractional Differential Equations

Following on from the work in Section 3.3 we now turn to the case where the derivative is fractional. We present Lubich's Fractional Multistep Method and the Fractional Adams Method for non-linear equations followed by the Diethelm-Chern Algorithm for linear FDEs.

As discussed in Chapter 2, FDEs can be interpreted as Volterra Integral Equations of the second kind. The kernel of the Volterra Integral Equation can be strongly singular requiring the integral to be evaluated as a Hadamard finite-part integral. The application of the fractional integral to the differential equation and merging of the initial conditions will provide an integral equation with a weakly singular kernel [20]. Both forms of Volterra Integral Equation will be required in the formation of the numerical methods described in subsequent sections.

Methods for Integral Equations

To assist later in this section a brief description is provided for two methods used to approximate integral equations: Product Rectangular Method and Product Trapezium Rule.

Product Rectangular Method

In the description of the Adams Method in Section 3.3.2 the rectangular methods (Forward and Backward Euler) were provided for ODEs. In the fractional case, and using the alternative form of FDEs Equation 2.6, the Volterra Integral Equation can be approximated using the *'Product Rectangular Method'* i.e.:

$$\begin{aligned}\int_a^T y(\tau)\,d\tau &= h\left(y_0 + y_1 + ... + y_{k-1} + y_k\right) \\ &= h\sum_{j=0}^{k} y_j \end{aligned} \tag{3.49}$$

where $y_j = y\left(a + j\left(\frac{T-a}{h}\right)\right)$.

The weights for the 'Product Rectangular Method' are obtained through the integration of the function $p_{rec} = 1$ with the kernel of the Volterra Integral Equation (see Section 3.4.2).

Product Trapezium Method

Again in Section 3.3.2 the Trapezium Rule for ODEs was provided. In the fractional case the 'Product Trapezium Rule' is another method used to approximate the Volterra Integral Equation element of equation 2.6. The *'Product Trapezium Rule'* can be summarised as:

$$\begin{aligned}\int_a^T y(\tau)\,d\tau &\approx h\left(\frac{1}{2}y_0 + y_1 + y_2 + \cdots + y_{n-2} + y_{n-1} + \frac{1}{2}y_n\right) \\ &\approx h\sum_{i=0}^{n} \omega_i y_i \end{aligned} \tag{3.50}$$

Where $y_j = y\left(a + j\left(\frac{T-a}{n}\right)\right)$ and the weights ω_i are obtained through the integration of the interpolating functions:

$$p_b = \frac{h-t}{h} \qquad \text{for } 0 \leq t \leq h \tag{3.51}$$

$$p_{int1} = \frac{t-(j-1)\,h}{h} \qquad \text{for } (j-1)\,h \leq t < jh \tag{3.52}$$

$$p_{int2} = 1 \qquad \text{for jh, singularity} \tag{3.53}$$

$$p_{int3} = \frac{(j+1)\,h-t}{h} \qquad \text{for } jh < t \leq (j+1)\,h \tag{3.54}$$

$$p_\epsilon = \frac{t-(k-1)\,h}{h} \qquad \text{for } (k-1)\,h \leq t \leq kh \tag{3.55}$$

The equations 3.51 - 3.55 are derived from the Lagrangian Interpolation Formula. Ralston et al [55] define the Lagrangian Interpolation Foruma as:

$$f(T) = \sum_{j=1}^{n} p_j f(t_j) + E(T) = \bar{y}(T) + E(T), \tag{3.56}$$

where

$$p_j = \frac{v_n(t)}{(T-t_j)\,v_n'(t_j)} \tag{3.57}$$

$$v_n'(t_j) = \frac{dv_n}{dT}|_{T=t_j} \tag{3.58}$$

$$v_n(T) = \prod_{i=1}^{n}(T-t_i), \tag{3.59}$$

and

$$E(T) = \frac{v_n(T)}{n!} f^{(n)}(\xi) \tag{3.60}$$

The integration of the interpolating functions in equations 3.51 - 3.55 provide the basis to the Fractional Adams-Moulton Method Section (3.4.2) and the Diethelm-Chern Algorithm (Section 3.4.3, see [60] and [18]).

3.4.1 Lubich's Fractional Multistep Method

Following on from Section 3.3.1 and described in [15] [41] [42] the Fractional Linear Multistep Method begins with the equivalent integral equation form of FDEs, Equation 2.6 where $a = 0$:

$$y(t) = y_0 + \frac{1}{\Gamma(\alpha)} \int_0^T (t-\tau)^{\alpha-1} f(\tau, y(\tau))\, d\tau \tag{3.61}$$

The convolution integral on the right hand side is approximated numerically using ODE Linear Multistep Methods principles. As discussed in Section 3.3.1 the First and Second Characteristic Polynomials can be used to form a function $\omega(z)$, the Taylor's Expansion of $\omega(z)$ provides the *'convolution weights'* ω_j for the quadrature. In the case of Fractional Multistep Methods the Taylor's Expansion is undertaken for $\Big(\omega(z)\Big)^{\alpha}$ where α is the fractional power. The equation to be evaluated is:

$$\Big(\omega(z)\Big)^{\alpha} = \left(\frac{\sigma\left(\frac{1}{z}\right)}{\rho\left(\frac{1}{z}\right)}\right)^{\alpha} = \sum_{j=0}^{\infty} \omega_j z^j \tag{3.62}$$

Considering an example where the underlying method is the Backward Euler Method $y_{n+1} - y_n = h f_{n+1}$ the First Characteristic Polynomial $\rho\left(\frac{1}{z}\right)$ remains the same as in the ODE case, see Equation 3.30 and repeated here for the Reader:

$$\rho\left(\frac{1}{z}\right) = \frac{1-z}{z}. \tag{3.63}$$

The Second Characteristic Polynomial $\sigma\left(\frac{1}{z}\right)$ again remains the same as the ODE case in Equation 3.31 and repeated here for the Reader:

$$\sigma\left(\frac{1}{z}\right) = \frac{1}{z}. \tag{3.64}$$

This leads to the same function $\omega(z)$ as Equation 3.32:

$$\omega(z) = (1-z)^{-1}$$

Performing the binomial expansion to the power α yields:

$$\begin{aligned}\left(\omega(z)\right)^{\alpha} &= \left((1-z)^{-1}\right)^{\alpha} \\ &= (1-z)^{-\alpha} \\ &= 1+(-\alpha)(-z)+\frac{(-\alpha)(-\alpha-1)(-z)^2}{2!}+\frac{(-\alpha)(-\alpha-1)(-\alpha-2)(-z)^3}{3!}+\dots \\ &= 1+\alpha z+\frac{\alpha(\alpha+1)z^2}{2!}+\frac{\alpha(\alpha+1)(\alpha+2)z^3}{3!}+\dots\end{aligned} \tag{3.65}$$

Therefore the convolution weights in this case would be:

$$\omega_0 = 1 \tag{3.66}$$

$$\omega_1 = \alpha \tag{3.67}$$

$$\omega_2 = \frac{\alpha(\alpha+1)}{2!} \tag{3.68}$$

$$\omega_3 = \frac{\alpha(\alpha+1)(\alpha+2)}{3!} \tag{3.69}$$

Starting Weights

A second set of weights called *'Starting Weights'* are formed to compensate for the asymptotic behaviour of the exact solution y near the origin [15]. This is the only violation of the convolution structure and is necessary for higher order FMMs [42]. This now means the integral equation element of equation 3.61 has the approximation:

$$\frac{1}{\Gamma(\alpha)}\int_0^{nh}(nh-\tau)^{(\alpha-1)}g(\tau)\,d\tau \approx h^{\alpha}\sum_{j=0}^{n}\omega_{n-j}g(jh)+h^{\alpha}\sum_{j=0}^{s}w_{nj}g(jh), \tag{3.70}$$

where $t=nh$, n is the total number of steps used to evaluate the integral, and h is the step size. The value of s is determined by the fractional derivative α and the order p of the underlying LMM used to obtain the convolution weights. In other words $s = card(A)-1$ where:

$$A=\left\{\gamma=j+l\alpha : j,l\in\{0,1,\dots\},\gamma\le p-1\right\}.$$

Equation 3.70 can be manipulated using the generating function $g(t)=t^{\gamma}$ to provide a linear system of equations used to determine the starting weights w_{nj} [15]. To illustrate this point an example will be considered.

Example $n = 1$

To begin, consider the left side of equation 3.70 which is the integral element. Using the substitution $r = nh - \tau$, therefore $\frac{dr}{d\tau} = -1$, and changing the limits of integration accordingly (including boundary change):

$$\begin{aligned} LHS &= \frac{1}{\Gamma(\alpha)} \left(\int_0^{nh} (nh - \tau)^{(\alpha-1)} g(\tau)\, d\tau \right) \\ &= \frac{1}{\Gamma(\alpha)} \int_{nh}^{0} r^{(\alpha-1)} g(nh - r)(-1)\, dr \\ &= \frac{-1}{\Gamma(\alpha)} \int_0^{nh} r^{(\alpha-1)} g(nh - r)(-1)\, dr \\ &= \frac{1}{\Gamma(\alpha)} \int_0^{nh} r^{(\alpha-1)} g(nh - r)\, dr \end{aligned}$$

Repeated integration by parts

$$
\begin{aligned}
LHS &= \frac{1}{\Gamma(\alpha)}\left(\left[g(nh-r)\frac{r^{\alpha}}{\alpha}\right]_0^{nh}+\int_0^{nh}\frac{r^{\alpha}}{\alpha}g^{'}(nh-r)\right)\\
&= \frac{1}{\Gamma(\alpha)}\left(\left[g(0)\frac{(nh)^{\alpha}}{\alpha}\right]-\left[g(nh)\frac{0^{\alpha}}{\alpha}\right]+\int_0^{nh}\frac{r^{\alpha}}{\alpha}g^{'}(nh-r)\right)\\
&= \frac{1}{\Gamma(\alpha)}\left(\left[g(0)\frac{(nh)^{\alpha}}{\alpha}\right]+\left(\left[g^{'}(nh-r)\frac{r^{(\alpha+1)}}{\alpha(\alpha+1)}\right]_0^{nh}+\int_0^{nh}\frac{r^{(\alpha+1)}}{\alpha(\alpha+1)}g^{''}(nh-r)\right)\right)\\
&= \frac{1}{\Gamma(\alpha)}\left(\left[g(0)\frac{(nh)^{\alpha}}{\alpha}\right]+\left[g^{'}(0)\frac{(nh)^{(\alpha+1)}}{\alpha(\alpha+1)}\right]-\left[g^{'}(nh)\frac{0^{(\alpha+1)}}{\alpha(\alpha+1)}\right]+\int_0^{nh}\frac{r^{(\alpha+1)}}{\alpha(\alpha+1)}g^{''}(nh-r)\right)\\
&= \frac{1}{\Gamma(\alpha)}\left(\left[g(0)\frac{(nh)^{\alpha}}{\alpha}\right]+\left[g^{'}(0)\frac{(nh)^{(\alpha+1)}}{\alpha(\alpha+1)}\right]+\left(\left[g^{''}(nh-r)\frac{r^{(\alpha+2)}}{\alpha(\alpha+1)(\alpha+2)}\right]_0^{nh}\right.\right.\\
&\quad \left.\left.+\int_0^{nh}\frac{r^{(\alpha+2)}}{\alpha(\alpha+1)(\alpha+2)}g^{'''}(nh-r)\right)\right)\\
&= \frac{1}{\Gamma(\alpha)}\left(\left[g(0)\frac{(nh)^{\alpha}}{\alpha}\right]+\left[g^{'}(0)\frac{(nh)^{(\alpha+1)}}{\alpha(\alpha+1)}\right]+\left[g^{''}(0)\frac{(nh)^{(\alpha+2)}}{\alpha(\alpha+1)(\alpha+2)}\right]\right.\\
&\quad \left.-\left[g^{''}(nh)\frac{0^{(\alpha+2)}}{\alpha(\alpha+1)(\alpha+2)}\right]+\int_0^{nh}\frac{r^{(\alpha+2)}}{\alpha(\alpha+1)(\alpha+2)}g^{'''}(nh-r)\right)\\
&= \frac{1}{\Gamma(\alpha)}\left(\left[g(0)\frac{(nh)^{\alpha}}{\alpha}\right]+\left[g^{'}(0)\frac{(nh)^{(\alpha+1)}}{\alpha(\alpha+1)}\right]+\left[g^{''}(0)\frac{(nh)^{(\alpha+2)}}{\alpha(\alpha+1)(\alpha+2)}\right]\right.\\
&\quad \left.+\left(\left[g^{'''}(nh-r)\frac{r^{(\alpha+3)}}{\alpha(\alpha+1)(\alpha+2)(\alpha+3)}\right]_0^{nh}+\int_0^{nh}\frac{r^{(\alpha+3)}}{\alpha(\alpha+1)(\alpha+2)(\alpha+3)}g^{IV}(nh-r)\right)\right)\\
&= \frac{1}{\Gamma(\alpha)}\left(\left[g(0)\frac{(nh)^{\alpha}}{\alpha}\right]+\left[g^{'}(0)\frac{(nh)^{(\alpha+1)}}{\alpha(\alpha+1)}\right]+\left[g^{''}(0)\frac{(nh)^{(\alpha+2)}}{\alpha(\alpha+1)(\alpha+2)}\right]+\left[g^{'''}(0)\frac{(nh)^{(\alpha+3)}}{\alpha(\alpha+1)(\alpha+2)(\alpha+3)}\right]\right.\\
&\quad \left.-\left[g^{'''}(nh)\frac{0^{(\alpha+3)}}{\alpha(\alpha+1)(\alpha+2)(\alpha+3)}\right]+\int_0^{nh}\frac{r^{(\alpha+3)}}{\alpha(\alpha+1)(\alpha+2)(\alpha+3)}g^{IV}(nh-r)\right)\\
&= \frac{1}{\Gamma(\alpha)}\left(\left[g(0)\frac{(nh)^{\alpha}}{\alpha}\right]+\left[g^{'}(0)\frac{(nh)^{(\alpha+1)}}{\alpha(\alpha+1)}\right]+\left[g^{''}(0)\frac{(nh)^{(\alpha+2)}}{\alpha(\alpha+1)(\alpha+2)}\right]\right.\\
&\quad \left.+\left[g^{'''}(0)\frac{(nh)^{(\alpha+3)}}{\alpha(\alpha+1)(\alpha+2)(\alpha+3)}\right]+\int_0^{nh}\frac{r^{(\alpha+3)}}{\alpha(\alpha+1)(\alpha+2)(\alpha+3)}g^{IV}(nh-r)\right). \qquad (3.71)
\end{aligned}
$$

Looking specifically at the case where $n = s = 1$ and $\alpha = 0.1$ results in the particular form of equation 3.71:

$$
\begin{aligned}
LHS &= \frac{1}{\Gamma(0.1)}\left(\left[g(0)\frac{(h)^{0.1}}{0.1}\right]+\left[g^{'}(0)\frac{(h)^{(1.1)}}{0.11}\right]+\left[g^{''}(0)\frac{(h)^{2.1}}{0.231}\right]\right.\\
&\quad \left.+\left[g^{'''}(0)\frac{(h)^{3.1}}{0.7161}\right]+\int_0^{h}\frac{r^{3.1}}{0.7161}g^{IV}(h-r)\right) \qquad (3.72)
\end{aligned}
$$

Returning to equation 3.70 and writing this for the specific example where $n = s = 1$ and $\alpha = 0.1$:

$$\begin{aligned}\frac{1}{\Gamma(0.1)}\int_0^h (h-\tau)^{(-0.9)}\, g(\tau)\, d\tau &\approx h^{0.1}\sum_{j=0}^{1}\omega_{1-j}g(jh) + h^{0.1}\sum_{j=0}^{1} w_{1j}g(jh) \\ &\approx h^{0.1}\Big(\omega_1 g(0) + \omega_0 g(h) + w_{10}g(0) + w_{11}g(h)\Big). \end{aligned} \tag{3.73}$$

Now considering the RHS of the equation 3.73 and using the generating function $g(t) = 1$

$$RHS = h^{0.1}\Big(\omega_1 + \omega_0 + w_{10} + w_{11}\Big) \tag{3.74}$$

Looking at the RHS of 3.73 again and using the generating function $g(t) = t^{0.1}$

$$\begin{aligned} RHS &= h^{0.1}\Big(\Big(\omega_1.0\Big) + \Big(\omega_0.\big(h^{0.1}\big)\Big) + \Big(w_{10}.0\Big) + \Big(w_{11}.h^{0.1}\Big)\Big) \\ &= h^{0.2}\omega_0 + h^{0.2}w_{11} \end{aligned} \tag{3.75}$$

Inputting the value of ω_0 from equation 3.66

$$RHS = \big(h^{0.2}.(1)\big) + \big(h^{0.2}.w_{11}\big)$$

Referring to equation 3.72, there is no equivalent coefficient for $h^{0.2}$ therefore:

$$\begin{aligned} 0 &= (1 + w_{11}) \\ w_{11} &= -1. \end{aligned} \tag{3.76}$$

Returning to equation 3.74 and inputting the results of equations 3.66 and 3.67 for ω_0 and ω_1 respectively:

$$\begin{aligned} RHS &= h^{0.1}\Big(\omega_1 + \omega_0 + w_{10} + w_{11}\Big) \\ &= h^{0.1}\Big(\alpha + 1 + w_{10} + (-1)\Big) \\ &= h^{0.1}\Big(\alpha + w_{10}\Big) \end{aligned}$$

Consulting equation 3.72 for equivalent LHS coefficient for $h^{0.1}$ and substituting value of α:

$$\begin{aligned} \frac{1}{\Gamma(0.1)}.\frac{1}{0.1} &= \alpha + w_{10} \\ \frac{10}{9.5135} - 0.1 &= w_{10} \\ w_{10} &= 0.9511. \end{aligned} \tag{3.77}$$

The resulting formula for the convolution integral would be:

$$\frac{1}{\Gamma(0.1)}\left(\int_0^h (h-\tau)^{(-0.9)} g(\tau)\, d\tau\right) \approx h^{0.1}\left(0.1g(0)+g(h)+0.9511g(0)-g(h)\right). \tag{3.78}$$

From the simple example above it can be seen that the starting weights can be calculated as a system and hence matrix form. This system is a Vandermonde Matrix which is ill-conditioned for large s and results in large relative errors due to cancellation of digits [42]. Diethelm et al [15] investigated methods for tackling the linear system of equations using Björck-Pereyra algorithm, standard and modified versions of Generalized Minimum Residual Method ($GMRes$), and LU decomposition. The use of the standard GMRes function produced the "best" results in calculating the starting weights.

General Formula for Lubich's Fractional Multistep Method

Once the starting and convolution weights have been obtained, the application of equation 3.70 to the original equation 2.6 yields the algorithm for Lubich's Fractional Mulitstep Methods:

$$y_n = y_0 + h^\alpha \sum_{j=0}^{n} \omega_{n-j} f(jh, y_j) + h^\alpha \sum_{j=0}^{s} w_{nj} f(jh, y_j) \tag{3.79}$$

The values for $y_0, y_1, ..., y_s$ will need to be obtained. These values could be obtained through an iterative scheme e.g. Newton's Method. Alternatively an explicit numerical method could be used for the solution at points $h, 2h, ..., sh$, in which case the FMM in equation 3.79 would begin when $s < k \leq n$ (k is the current step where $1 \leq k \leq n$). Irrespective of how the values $y_0, y_1, ..., y_s$ are obtained the Equation 3.79 remains implicit. In the

application of this method in Chapter 10 an alternative numerical method is used to 'predict' the value at y_n which was used subsequently in Lubich's FMM algorithm. This was a variation to the implementation of Lubich's FMM in the paper by Diethelm et al [15] where Newton's Iterative Method was applied.

3.4.2 Fractional Adams Method

Building on the work of the general Adams Methods in Section 3.3.2, and the Predictor-Corrector Section 3.3.3, we now present the Fractional Adams Method [18] [20]. The Fractional Adams Method, also referred to as 'Adams-Bashforth-Moulton Method, uses the fractional versions of the Adams-Moulton Method and Adams-Bashforth Method as a 'Predictor' - 'Corrector' pair. To begin let us consider the methods separately.

Fractional Adams-Moulton Method

In [4] and [20] the Fractional Adams-Moulton Method is derived using the 'Product Trapezium Rule'. As discussed in 3.49 the 'Product Trapezium Rule' replaces the Volterra Integral Equation in the alternative definition for an FDE Equation 2.6. The weights $\omega_{j,k}$ are derived by integrating the interpolating functions 3.51 - 3.55.

Considering the Volterra Integral Equation only:

$$I = \int_a^T (t-\tau)^{\alpha-1} f(\tau, y(\tau))\, d\tau. \tag{3.80}$$

The function $f(\tau, y(\tau))$ will be replaced by one of the integrating functions 3.51 - 3.55. The interval of integration $[a, T]$ will be altered to reflect the interpolating function. Additionally t will be replaced by its alternative meaning $t = kh$.

Internal Weights

To begin let us consider the internal weights at point $(j-1) \leq t \leq j$ and corresponding interpolating function 3.52. Equation 3.80 becomes:

$$I_{j,(j-1)} = \int_{(j-1)h}^{jh} \frac{\tau - (j-1)\,h}{h} . \frac{1}{(kh-\tau)^{1-\alpha}} d\tau.$$

Using substitution $\tau = (k-s)\,h \rightarrow du = (-h)\,ds$ and changing the limits of integration accordingly:

$$\begin{aligned}
I_{j,(j-1)} &= \int_{s((j-1)h)}^{s(jh)} \frac{(k-s)\,h-(j-1)\,h}{h} \frac{1}{(kh-(k-s)\,h)^{1-\alpha}}\,(-h)\,ds \\
&= (-1)\,(-h)\int_{k-j}^{k-j+1} \frac{k-s-j+1}{(sh)^{1-\alpha}} ds \\
&= (h)\int_{k-j}^{k-j+1} \frac{k-(j-1)-s}{h.h^{-\alpha}.s^{1-\alpha}} ds \\
&= h^{\alpha}\int_{k-j}^{k-j+1} (k-(j-1))\,s^{\alpha-1} - s^{\alpha} ds \\
&= h^{\alpha}\left[\frac{(k-(j-1))\,s^{\alpha}}{\alpha} - \frac{s^{\alpha+1}}{\alpha+1}\right]_{k-j}^{k-j+1} \\
&= h^{\alpha}\left(\left[\frac{(k-(j-1))\,(k-(j-1))^{\alpha}}{\alpha} - \frac{(k-(j-1))^{\alpha+1}}{\alpha+1}\right] - \left[\frac{(k-(j-1))\,(k-j)^{\alpha}}{\alpha} - \frac{(k-j)^{\alpha+1}}{\alpha+1}\right]\right) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big((\alpha+1)\,(k-(j-1))^{\alpha+1} - \alpha\,(k-(j-1))^{\alpha+1} - (\alpha+1)\,(k-(j-1))\,(k-j)^{\alpha} + \alpha\,(k-j)^{\alpha+1}\Big) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big(\alpha\,(k-(j-1))^{\alpha+1} + (k-(j-1))^{\alpha+1} - \alpha\,(k-(j-1))^{\alpha+1} - \alpha k\,(k-j)^{\alpha} + \alpha j\,(k-j)^{\alpha} \\
&\quad -\alpha\,(k-j)^{\alpha} - k\,(k-j)^{\alpha} + j\,(k-j)^{\alpha} - (k-j)^{\alpha} + \alpha\,(k-j)^{\alpha+1}\Big) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big((k-(j-1))^{\alpha+1} - \alpha k\,(k-j)^{\alpha} + \alpha j\,(k-j)^{\alpha} - (k-j)\,(k-j)^{\alpha} + \alpha k\,(k-j)^{\alpha} \\
&\quad -\alpha j\,(k-j)^{\alpha} - \alpha\,(k-j)^{\alpha} - (k-j)^{\alpha}\Big) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big((k-(j-1))^{\alpha+1} - (k-j)^{\alpha+1} - \alpha\,(k-j)^{\alpha} - (k-j)^{\alpha}\Big). \qquad (3.81)
\end{aligned}$$

Returning to equation 3.80 with interpolating function 3.54 results in the equation:

$$I_{(j+1),j} = \int_{jh}^{(j+1)h} \frac{(j+1)\,h-\tau}{h} . \frac{1}{(kh-\tau)^{1-\alpha}} d\tau.$$

Using the substiution $\tau = (k-s)\,h \rightarrow du = (-h)\,ds$

$$\begin{aligned}
I_{(j+1),j} &= \int_{s(jh)}^{s((j+1)h)} \frac{(j+1)\,h-(k-s)\,h}{h} . \frac{1}{(kh-kh+sh)^{\alpha+1}}\,(-h)\,ds \\
&= (-h)\int_{k-j}^{k-(j+1)} \frac{j+1-k+s}{(sh)^{1-\alpha}} ds. \qquad (3.82)
\end{aligned}$$

Changing order of integration:

$$
\begin{aligned}
I_{(j+1),j} &= (h)\int_{k-(j+1)}^{k-j} \frac{s-(k-(j+1))}{h.h^{-\alpha}.s^{1-\alpha}}ds \\
&= h^{\alpha}\int_{k-(j+1)}^{k-j} s^{\alpha}-(k-(j+1))\,s^{\alpha-1}ds \\
&= h^{\alpha}\left[\frac{s^{\alpha+1}}{\alpha+1}-\frac{(k-(j+1))\,s^{\alpha}}{\alpha}\right]_{k-(j+1)}^{k-j} \\
&= h^{\alpha}\left(\left[\frac{(k-j)^{\alpha+1}}{\alpha+1}-\frac{(k-(j+1))\,(k-j)^{\alpha}}{\alpha}\right]-\left[\frac{(k-(j+1))^{\alpha+1}}{\alpha+1}-\frac{(k-(j+1))\,(k-(j+1))^{\alpha}}{\alpha}\right]\right) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big(\alpha\,(k-j)^{\alpha+1}-(\alpha+1)\,(k-(j+1))\,(k-j)^{\alpha}-\alpha\,(k-(j+1))^{\alpha+1}+(\alpha+1)\,(k-(j+1))^{\alpha+1}\Big) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big(\alpha\,(k-j)^{\alpha+1}-\alpha k\,(k-j)^{\alpha}+\alpha j\,(k-j)^{\alpha}+\alpha\,(k-j)^{\alpha}-k\,(k-j)^{\alpha}+j\,(k-j)^{\alpha} \\
&\quad +(k-j)^{\alpha}-\alpha\,(k-(j+1))^{\alpha+1}+\alpha\,(k-(j+1))^{\alpha+1}+(k-(j+1))^{\alpha+1}\Big) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big(\alpha\,(k-j)^{\alpha+1}-\alpha\,(k-j)\,(k-j)^{\alpha}+\alpha\,(k-j)^{\alpha}-(k-j)\,(k-j)^{\alpha}+(k-j)^{\alpha}+(k-(j+1))^{\alpha+1}\Big) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big(\alpha\,(k-j)^{\alpha+1}-\alpha\,(k-j)^{\alpha+1}+\alpha\,(k-j)^{\alpha}-(k-j)^{\alpha+1}+(k-j)^{\alpha}+(k-(j+1))^{\alpha+1}\Big) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big(\alpha\,(k-j)^{\alpha}-(k-j)^{\alpha+1}+(k-j)^{\alpha}+(k-(j+1))^{\alpha+1}\Big). \qquad (3.83)
\end{aligned}
$$

Adding equations 3.81 and 3.83 together:

$$
\begin{aligned}
I_{int} &= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big((k-(j-1))^{\alpha+1}-(k-j)^{\alpha+1}-\alpha\,(k-j)^{\alpha}-(k-j)^{\alpha} \\
&\quad +\alpha\,(k-j)^{\alpha}-(k-j)^{\alpha+1}+(k-j)^{\alpha}+(k-(j+1))^{\alpha+1}\Big) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big((k-j+1)^{\alpha+1}+(k-j-1)^{\alpha+1}-2\,(k-j)^{\alpha+1}\Big). \qquad (3.84)
\end{aligned}
$$

End Weights

Returning to equation 3.80 with the interpolating function 3.55:

$$I_{end}=\int_{(j-1)h}^{jh}\frac{\tau-(j-1)\,h}{h}.\frac{1}{(jh-\tau)^{1-\alpha}}d\tau.$$

Using the substitution $\tau=(j-s)\,h\rightarrow du=(-h)\,ds$

$$\begin{aligned}
I_{end} &= \int_{s((j-1)h)}^{s(jh)} \frac{(j-s)\,h-(j-1)\,h}{h}.\frac{1}{(jh-jh+sh)^{1-\alpha}}\,(-h)\,ds \\
&= (-h)\int_1^0 \frac{j-s-j+1}{h.h^{-\alpha}.s^{1-\alpha}}ds \\
&= (h)\int_0^1 \frac{1-s}{h.h^{-\alpha}.s^{1-\alpha}}ds \\
&= (h^\alpha)\int_0^1 s^{\alpha-1}-s^\alpha ds \\
&= (h^\alpha)\left[\frac{s^\alpha}{\alpha}-\frac{s^{\alpha+1}}{\alpha+1}\right]_0^1 \\
&= (h^\alpha)\left(\left[\frac{1}{\alpha}-\frac{1}{\alpha+1}\right]-\left[0-0\right]\right).
\end{aligned} \tag{3.85}$$

Rearranging the equation now gives the formula:

$$I_{end} = \frac{h^\alpha}{\alpha\,(\alpha+1)}. \tag{3.86}$$

Weight at origin

Now considering the case of the weight at the origin. Recalling the integral equation 3.80 with the interpolating function 3.51 yields:

$$I_{origin} = \int_0^h \frac{\tau-h}{h}.\frac{1}{(kh-\tau)^{1-\alpha}}d\tau.$$

Using the substitution $\tau = (k-s)\,h \rightarrow du = (-h)\,ds$

$$
\begin{aligned}
I_{origin} &= \int_{s(0)}^{s(h)} \frac{h-(k-s)\,h}{h}.\frac{1}{(kh-kh+sh)^{1-\alpha}}\,(-h)\,ds \\
&= (-h)\int_{k}^{k-1} \frac{1-k+s}{h.h^{-\alpha}.s^{1-\alpha}}ds \\
&= \int_{k-1}^{k} \frac{s-(k-1)}{h^{-\alpha}.s^{1-\alpha}}ds \\
&= h^{\alpha}\int_{k-1}^{k} s^{\alpha}-(k-1)\,s^{\alpha-1}ds \\
&= h^{\alpha}\left[\frac{s^{\alpha+1}}{\alpha+1}-\frac{(k-1)\,s^{\alpha}}{\alpha}\right]_{k-1}^{k} \\
&= h^{\alpha}\left(\left[\frac{k^{\alpha+1}}{\alpha+1}-\frac{(k-1)\,k^{\alpha}}{\alpha}\right]-\left[\frac{(k-1)^{\alpha+1}}{\alpha+1}-\frac{(k-1)\,(k-1)^{\alpha}}{\alpha}\right]\right) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\left(\alpha k^{\alpha+1}-(\alpha+1)\,(k-1)\,k^{\alpha}-\alpha\,(k-1)^{\alpha+1}+(\alpha+1)\,(k-1)^{\alpha+1}\right) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\Big(\alpha k^{\alpha+1}-\alpha k^{\alpha+1}+\alpha k^{\alpha}-k^{\alpha+1}+k^{\alpha} \\
&\quad -\alpha\,(k-1)^{\alpha+1}+\alpha k\,(k-1)^{\alpha}-\alpha\,(k-1)^{\alpha}+k\,(k-1)^{\alpha}-(k-1)^{\alpha}\Big) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\left(\alpha k^{\alpha}-k^{\alpha+1}+k^{\alpha}-\alpha k\,(k-1)^{\alpha}+\alpha\,(k-1)^{\alpha}+\alpha k\,(k-1)^{\alpha}-\alpha\,(k-1)^{\alpha}+k\,(k-1)^{\alpha}-(k-1)^{\alpha}\right) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\left((k-1)^{\alpha}\,(k-1)-(k-\alpha-1)\,k^{\alpha}\right) \\
&= \frac{h^{\alpha}}{\alpha\,(\alpha+1)}\left((k-1)^{\alpha+1}-(k-\alpha-1)\,k^{\alpha}\right). \qquad (3.87)
\end{aligned}
$$

Summary of weight equations

Returning to the Volterra Integral Equation 3.61 and repeated here:

$$y\,(t)=y_0+\frac{1}{\Gamma\,(\alpha)}\int_0^T (t-\tau)^{\alpha-1}\,f\,(t,y\,(\tau))\,d\tau. \qquad (3.88)$$

In 3.88 the coefficient of the integral is $\frac{1}{\Gamma(\alpha)}$. The coefficient of the calculated weight equations 3.84 - 3.87 is $\frac{h^{\alpha}}{\alpha(\alpha+1)}$. Combining these coefficients together into the variable $X_{FAMoulton}$:

$$
\begin{aligned}
X_{FAMoulton} &= \frac{1}{\Gamma\,(\alpha)}.\frac{h^{\alpha}}{\alpha\,(\alpha+1)} \\
&= \frac{h^{\alpha}}{(\alpha+1)\,\alpha\Gamma\,(\alpha)}.
\end{aligned}
$$

Using the alternative identity of $\Gamma\,(\alpha)$

$$\begin{aligned} X_{FAMoulton} &= \frac{h^{\alpha}}{(\alpha+1)\,\alpha\,(\alpha-1)\,!} \\ &= \frac{h^{\alpha}}{(\alpha+1)\,!} \\ &= \frac{h^{\alpha}}{\Gamma(\alpha+2)}. \end{aligned} \tag{3.89}$$

Following the integration of the interpolating functions 3.84 - 3.87, and the change to the coefficient 3.89, we are now in a position to summarise the weight equations for the Fractional Adams-Moulton Method as:

$$b_{j,k} = \frac{1}{\Gamma(2+\alpha)} \begin{cases} \left((k-1)^{1+\alpha} - (k-\alpha-1)\,k^{\alpha} \right) & \text{when j=0} \\ \left((k-j+1)^{1+\alpha} + (k-j-1)^{1+\alpha} - 2\,(k-j)^{1+\alpha} \right. & \text{when } 1 \leq j \leq k-1 \\ 1 & \text{when j=k} \end{cases} \tag{3.90}$$

Fractional Adams-Moulton Method Algorithm

The algorithm for the Fractional Adams-Moulton Method is provided in [4] as:

$$y_k = \sum_{j=0}^{\lceil \alpha \rceil - 1} \frac{t_k^j}{j!} y_0^{(j)} + h^{\alpha} \left(\sum_{j=0}^{k-1} b_{j,k} f\left(t_j, y_j\right) + b_{k,k} f\left(t_k, y_k^p\right) \right). \tag{3.91}$$

As in the general Fractional Multistep Methods, the integral equation has been replaced by quadrature. The end weight at $b_{k,k}$ has been taken out of the characteristic polynomial to highlight the need for a 'predictor' to the non-linear term. Unlike the general Fractional Multistep Methods, no starting weights are required to enable high order methods.

The 'prediction' of the y_k^P is undertaken by the Fractional Adams-Bashforth Method.

Fractional Adams-Bashforth Method

As indicated in [4] the Fractional Adams-Bashforth Method weights are obtained in a similar manner to the Fractional Adams-Moulton weights. Instead of the Product Trapezium Method, the Product Rectangular Method

is employed to obtain the weight equation. By integrating 3.80 with the interpolating function $p_{rec} = 1$ we obtain the weights $b^*_{j,k}$ for the Method:

$$I_{FAB} = \int_{jh}^{(j+1)h} (kh - \tau)^{\alpha-1} \, d\tau. \tag{3.92}$$

Using the substitution $\tau = (k - s)\, h \rightarrow du = (-h)$

$$I_{FAB} = \int_{s(jh)}^{s((j+1)h)} (kh - (k - s)\, h)^{\alpha-1} \, (-h) \, ds.$$

Changing limits of integration

$$\begin{aligned}
I_{FAB} &= (-h) \int_{k-j}^{k-1-j} (kh - kh + sh)^{\alpha-1} \, ds \\
&= h \int_{k-1-j}^{k-j} (sh)^{\alpha-1} \, ds \\
&= h \int_{k-1-j}^{k-j} h^{-1}.h^{\alpha}.s^{\alpha-1} ds \\
&= h^{\alpha} \int_{k-1-j}^{k-j} s^{\alpha-1} ds \\
&= h^{\alpha} \left[\frac{s^{\alpha}}{\alpha} \right]_{k-1-j}^{k-j} \\
&= h^{\alpha} \left(\left[\frac{(k-j)^{\alpha}}{\alpha} \right] - \left[\frac{(k-1-j)^{\alpha}}{\alpha} \right] \right) \\
&= \frac{h^{\alpha}}{\alpha} \left((k-j)^{\alpha} - (k-1-j)^{\alpha} \right).
\end{aligned} \tag{3.93}$$

As with the Fractional Adams-Moulton Method some manipulation of the coefficient $\frac{1}{\Gamma(\alpha)}$ (integral equation 3.88) and $\frac{h^{\alpha}}{\alpha}$ (the weight equation 3.93) is required. This will be calculated to variable $X_{FABashforth}$:

$$\begin{aligned}
X_{FABashforth} &= \frac{1}{\Gamma(\alpha)}.\frac{h^{\alpha}}{\alpha} \\
&= \frac{h^{\alpha}}{\alpha\Gamma(\alpha)}.
\end{aligned}$$

Using the alternative identity for $\Gamma(\alpha)$:

$$\begin{aligned} X_{FABashforth} &= \frac{h^{\alpha}}{\alpha(\alpha-1)!} \\ &= \frac{h^{\alpha}}{(\alpha)!} \\ &= \frac{h^{\alpha}}{\Gamma(\alpha+1)}. \end{aligned} \tag{3.94}$$

We now have the equation for the weights $b^*_{j,k}$ by combining the result of equations 3.93 and 3.94. To summarise:

$$b^*_{j,k} = \frac{(k-j)^{\alpha} - (k-1-j)^{\alpha}}{\Gamma(\alpha+1)}. \tag{3.95}$$

Fractional Adams-Bashforth Method Algorithm

In [4] and [14] the algorithm is provided for the Fractional Adams-Moulton Method:

$$y^p_k = \sum_{j=0}^{\lceil\alpha\rceil-1} \frac{t^j_k}{j!} y^{(j)}_0 + h^{\alpha} \sum_{j=0}^{k-1} b^*_{j,k} f(t_j, y_j), \tag{3.96}$$

where all previous values $y_0, y_1, \ldots, y_j$ have already been calculated. As with the Fractional Adams-Moulton Method the integral equation has been replaced with quadrature.

Final Comment

As discussed in the introduction to this section. The Fractional Adams-Bashforth and Fractional Adams-Moulton Methods form a predictor-corrector pair which is called the *'Fractional Adams Method'*. The application of the 'Corrector' element of the Fractional Adams Method can be applied multiple times however the subsequent programming and application only consider the case where the procedure occurs once. This mirrors Diethelm's application in [14] and enables comparison of the computer program performance.

Fractional Adams Method: An Example

k	**Predictor** y_k^P	**Final** y_k
0	-	1
1	0.652	0.705
2	0.246	0.388
3	-0.089	0.147
4	-0.326	-0.007
5	-0.483	-0.091
6	-0.563	-0.126
7	-0.603	-0.129
8	-0.617	-0.114
9	-0.617	-0.094
10	-0.613	-0.073

Tab. 3.2: Results of the FAM Example

j	$a_{j,10}$	$b_{j,10}$
0	1.100	2.189
1	2.154	2.117
2	2.079	2.039
3	1.997	1.953
4	1.906	1.858
5	1.805	1.749
6	1.687	1.621
7	1.546	1.465
8	1.366	1.253
9	1.090	0.857

Tab. 3.3: Results of the FAM Example (2)

By way of an example, consider the following equation:

$$D_0^{1.3} y(x) = -y(x) \tag{3.97}$$

With initial condition data $y_0 = 1$ and $y^{(k)}(0) = 0$. Considering the case where the termination point, $T = 5$ and the step size is $h = 0.5$. Results for this example are given in Tables 3.2 and 3.3. For the sake of brevity, the results for the weights $a_{j,k}$ and $b_{j,k}$ are for the final step $k = 10$.

3.4.3 Diethelm-Chern Algorithm

In Deithelm [13] (see also [60] and [5]) an algorithm for fractional differential equations is provided for linear FDEs. This Algorithm is termed Diethelm-

Chern in this thesis to reference the work of Chern see [19] and references there-in. Diethelm considers initial value problems of the form:

$$D^{\alpha}\left[y-y_0\right](t)=\beta y(t)+f(t), \tag{3.98}$$

where $0 \leq t \leq 1$

In 3.98 the intial condition y_0 has been incorporated into the FDE. Here D^{α} represents the fractional derivative of order $0<\alpha<1$. The restriction on the values of t is not essential. Also please note that the coefficient $\beta \leq 0$.

The algorithm begins with the Riemann-Liouville fractional derivative which is provided in Chapter 2 and repeated here for the Reader:

$$\left(D^{\alpha}y\right)(t)=\frac{1}{\Gamma(1-\alpha)}\frac{d}{dt}\int_0^T\frac{y(\tau)}{(t-\tau)^{\alpha}}d\tau.$$

where $a=0$. By interchanging differentiation and integration the Riemann-Liouville fractional derivative is transformed into the form:

$$\left(D^{\alpha}y\right)(t)=\frac{1}{\Gamma(-\alpha)}\int_0^T\frac{y(\tau)}{(t-\tau)^{1+\alpha}}d\tau, \tag{3.99}$$

where the integral equation has to be interpreted using the Hadamard finite-part integral.

Integration of Interpolating Functions

The weights and algorithm for the Diethelm-Chern Method are obtained in a similar manner to the Fractional Adams-Moulton Method (using the Product Trapezium Method).

To begin, the integral equation of 3.99 is repeated here and assigned to variable I_{DC}:

$$I_{DC}=\int_0^T\frac{y(\tau)}{(t-\tau)^{1+\alpha}}. \tag{3.100}$$

The function $y(\tau)$ in equation 3.100 is replaced by the interpolating functions from Section 3.4. The resultant integral equation is evaluated between the interpolation points. Please note: following the initial condition information in 3.99, $T = kh = 1$.

Interior Interpolating Functions

To begin, consider the integral equation 3.100 where the function is replaced with the interpolating function Equation 3.52:

$$\begin{aligned} I_{int1} &= \int_{(j-1)h}^{jh} \frac{(u-(j-1)h)}{h(kh-u)^{1+\alpha}} du \\ &= \frac{1}{h}\int_{(j-1)h}^{jh} \frac{(u-(j-1)h)}{(kh-u)^{1+\alpha}} du. \end{aligned}$$

Applying the substitution $u = (k-s)h \Rightarrow du = -hds$

$$I_{int1} = \frac{1}{h}\int_{u((j-1)h)}^{u(jh)} \frac{((k-s)h-(j-1)h)}{(sh)^{1+\alpha}} (-h)\, ds.$$

Changing the limits of integration:

$$\begin{aligned} I_{int1} &= \int_{k-(j-1)}^{k-j} \frac{((k-s)h-(j-1)h)}{h.h^{\alpha}.s^{1+\alpha}} ds \\ &= h^{-\alpha}\int_{k-(j-1)}^{k-j} \frac{k-s-(j-1)}{s^{1+\alpha}} ds \\ &= h^{-\alpha}\int_{k-(j-1)}^{k-j} (k-(j-1))\, s^{-1-\alpha} - s^{-\alpha} ds \\ &= h^{-\alpha}\left[\frac{k-(j-1)\, s^{-\alpha}}{-\alpha} - \frac{s^{-\alpha+1}}{-\alpha+1}\right]_{k-j}^{k-(j-1)} \\ &= h^{-\alpha}\left[\frac{k-(j-1)}{-\alpha s^{\alpha}} - \frac{s^{1-\alpha}}{1-\alpha}\right]_{k-j}^{j-(j-1)} \\ &= h^{-\alpha}\left(\frac{k-(j-1)}{-\alpha(k-(j-1))^{\alpha}} - \frac{(k-(j-1))^{1-\alpha}}{1-\alpha} + \frac{k-(j-1)}{\alpha(k-j)^{\alpha}} + \frac{(k-j)^{1-\alpha}}{1-\alpha}\right) \\ &= h^{-\alpha}\left(\frac{(k-(j-1))^{1-\alpha}}{-\alpha} - \frac{(k-(j-1))^{1-\alpha}}{1-\alpha} + \frac{k-(j-1)}{\alpha(k-j)^{\alpha}} + \frac{(k-j)^{1-\alpha}}{1-\alpha}\right). \end{aligned} \tag{3.101}$$

Considering the integral equation 3.100 with the interpolating function 3.54:

$$I_{int3} = \int_{jh}^{(j+1)h} \frac{(j+1)h - u}{h(kh-u)^{1+\alpha}} du$$
$$= \frac{1}{h}\int_{jh}^{(j+1)h} \frac{(j+1)h - u}{h(kh-u)^{1+\alpha}} du.$$

Applying the substitution $u = (k-s)h \Rightarrow du = -hds$

$$I_{int3} = \frac{1}{h}\int_{u(jh)}^{u((j+1)h)} \frac{((j+1)h - (k-s)h)}{(kh-(k-s)h)^{1+\alpha}} (-h)\, ds.$$

Changing the limits of integration:

$$\begin{aligned}
I_{int3} &= (-1)\int_{k-j}^{k-(j+1)} \frac{((j+1)h-(k-s)h)}{(kh-kh+sh)^{1+\alpha}} ds \\
&= \int_{k-(j+1)}^{k-j} \frac{((j+1)h-(k-s)h)}{h.h^{\alpha}.s^{1+\alpha}} ds \\
&= h^{-\alpha}\int_{k-(j+1)}^{k-j} \frac{(j+1)-k}{s^{1+\alpha}} + \frac{s}{s^{1+\alpha}} ds \\
&= (-h)^{-\alpha}\int_{k-(j+1)}^{k-j} (k-(j+1))\, s^{-1-\alpha} - s^{-\alpha} ds \\
&= (-h)^{-\alpha}\left[\frac{(k-(j+1))\, s^{-\alpha}}{-\alpha} - \frac{s^{-\alpha+1}}{-\alpha+1}\right]_{k-(j+1)}^{k-j} \\
&= (-h)^{-\alpha}\left(\left[\frac{(k-(j+1))}{-\alpha(k-j)^{\alpha}} - \frac{(k-j)^{1-\alpha}}{1-\alpha}\right] - \left[\frac{(k-(j+1))}{-\alpha(k-(j+1))^{\alpha}} - \frac{(k-(j+1))^{1+\alpha}}{1-\alpha}\right]\right) \\
&= (-h)^{-\alpha}\left(\frac{(k-(j+1))}{-\alpha(k-j)^{\alpha}} - \frac{(k-j)^{1-\alpha}}{1-\alpha} + \frac{(k-(j+1))^{1-\alpha}}{\alpha} + \frac{(k-(j+1))^{1-\alpha}}{1-\alpha}\right) \quad (3.102)
\end{aligned}$$

Combining equations 3.101 and 3.102 together:

$$\begin{aligned}
I_{int} &= I_{int1} + I_{int3} \\
&= h^{-\alpha}\left(\frac{(k-(j-1))^{1-\alpha}}{-\alpha} - \frac{(k-(j-1))^{1-\alpha}}{1-\alpha} + \frac{k-(j-1)}{\alpha(k-j)^{\alpha}} + \frac{(k-j)^{1-\alpha}}{1-\alpha}\right) \\
&\quad -h^{-\alpha}\left(\frac{(k-(j+1))}{-\alpha(k-j)^{\alpha}} - \frac{(k-j)^{1-\alpha}}{1-\alpha} + \frac{(k-(j+1))^{1-\alpha}}{\alpha} + \frac{(k-(j+1))^{1-\alpha}}{1-\alpha}\right) \\
&= \frac{h^{-\alpha}(k-(j-1))^{1-\alpha}}{-\alpha} - \frac{h^{-\alpha}(k-(j-1))^{1-\alpha}}{1-\alpha} + \frac{h^{-\alpha}(k-(j-1))}{\alpha(k-j)^{\alpha}} \\
&\quad + \frac{h^{-\alpha}(k-j)^{1-\alpha}}{1-\alpha} + \frac{h^{-\alpha}(k-(j+1))}{\alpha(k-j)^{\alpha}} + \frac{h^{-\alpha}(k-j)^{1-\alpha}}{1-\alpha} \\
&\quad - \frac{h^{-\alpha}(k-(j+1))^{1-\alpha}}{\alpha} - \frac{h^{-\alpha}(k-(j+1))^{1-\alpha}}{1-\alpha}
\end{aligned}$$

$$\begin{aligned}
I_{int} &= \frac{2\left(h\right)^{-\alpha}\left(k-j\right)^{1-\alpha}}{1-\alpha} - \frac{\left(h^{-\alpha}\left(k-(j+1)\right)^{1-\alpha}\right)\left(1-\alpha\right)+\alpha h^{-\alpha}\left(k-(j+1)\right)^{1-\alpha}}{\alpha\left(1-\alpha\right)} \\
&\quad - \frac{\left(h^{-\alpha}\left(k-(j-1)\right)^{1-\alpha}\right)\left(1-\alpha\right)+\alpha h^{-\alpha}\left(k-(j-1)\right)^{1-\alpha}}{\alpha\left(1-\alpha\right)} + \frac{h^{\alpha}\left(k-(j-1)+k-(j+1)\right)}{\alpha\left(k-j\right)^{\alpha}} \\
&= \frac{2h^{-\alpha}\left(k-j\right)^{1-\alpha}}{1-\alpha} - \frac{\left(h^{-\alpha}\left(k-(j+1)\right)^{1-\alpha}-\alpha h^{-\alpha}\left(k-(j+1)\right)^{1-\alpha}+\alpha h^{-\alpha}\left(k-(j+1)\right)^{1-\alpha}\right)}{\alpha\left(1-\alpha\right)} \\
&\quad - \frac{\left(h^{-\alpha}\left(k-(j-1)\right)^{1-\alpha}-\alpha h^{-\alpha}\left(k-(j-1)\right)^{1-\alpha}+\alpha h^{-\alpha}\left(k-(j-1)\right)^{1-\alpha}\right)}{\alpha\left(1-\alpha\right)} + \frac{2h^{-\alpha}\left(k-j\right)}{\alpha\left(k-j\right)^{\alpha}} \\
&= \left[\frac{\alpha 2h^{-\alpha}\left(k-j\right)^{1-\alpha}+\left(1-\alpha\right)\left(2h^{-\alpha}\left(k-j\right)^{1-\alpha}\right)}{\alpha\left(1-\alpha\right)}\right] - \frac{h^{-\alpha}\left(k-(j+1)\right)^{1-\alpha}}{\alpha\left(1-\alpha\right)} - \frac{h^{-\alpha}\left(k-(j-1)\right)^{1-\alpha}}{\alpha\left(1-\alpha\right)} \\
&= \frac{2h^{-\alpha}\left(k-j\right)^{1-\alpha}}{\alpha\left(1-\alpha\right)} - \frac{h^{-\alpha}\left(k-(j+1)\right)^{1-\alpha}}{\alpha\left(1-\alpha\right)} - \frac{h^{-\alpha}\left(k-(j-1)\right)^{1-\alpha}}{\alpha\left(1-\alpha\right)}. \qquad (3.103)
\end{aligned}$$

Rearranging results in equation 3.104 for internal weights:

$$\alpha\left(1-\alpha\right)h^{\alpha}I_{int} = 2\left(k-j\right)^{1-\alpha} - \left(k-(j-1)\right)^{1-\alpha} - \left(k-(j+1)\right)^{1-\alpha}. \quad (3.104)$$

Initial Interpolating Function

Now considering the integral equation 3.100 with the interpolating function 3.51:

$$\begin{aligned}
I_{intb} &= \int_0^h \frac{h-u}{h\left(kh-u\right)^{1+\alpha}}du \\
&= \frac{1}{h}\int_0^h \frac{h-u}{\left(kh-u\right)^{1+\alpha}}du.
\end{aligned}$$

Applying the substitution $u = \left(k-s\right)h \Rightarrow du = -hds$

$$I_{intb} = \frac{1}{h}\int_{u(0)}^{u(h)} \frac{h-\left(k-s\right)h}{\left(kh-\left(k-s\right)h\right)^{1+\alpha}}\left(-h\right)ds.$$

Changing limits of integration:

$$
\begin{aligned}
I_{intb} &= \frac{1}{h}\int_{k}^{k-1}\frac{h-(k-s)\,h}{(kh-(k-s)\,h)^{1+\alpha}}\,(-h)\,ds \\
&= \int_{k-1}^{k}\frac{h-(k-s)\,h}{(kh-(k-s)\,h)^{1+\alpha}}ds \\
&= \int_{k-1}^{k}\frac{h-(k-s)\,h}{(sh)^{1+\alpha}}ds \\
&= \int_{k-1}^{k}\frac{h-(k-s)\,h}{h.h^{\alpha}.s^{1+\alpha}}ds \\
&= h^{-\alpha}\int_{k-1}^{k}\frac{1-(k-s)}{s^{1+\alpha}}ds \\
&= h^{-\alpha}\int_{k-1}^{k}\frac{s-(k-1)}{s^{1+\alpha}}ds \\
&= h^{-\alpha}\int_{k-1}^{k}\frac{s}{s^{1+\alpha}}-\frac{(k-1)}{s^{1+\alpha}}ds \\
&= h^{-\alpha}\int_{k-1}^{k}s^{-\alpha}-(k-1)\,u^{-1-\alpha}ds \\
&= h^{-\alpha}\left[\frac{s^{-\alpha+1}}{-\alpha+1}-\frac{(k-1)\,s^{-\alpha}}{-\alpha}\right]_{k-1}^{k} \\
&= h^{-\alpha}\left[\frac{k^{1-\alpha}}{1-\alpha}+(k-1)\,k^{-\alpha}\alpha\right]-\left[\frac{(k-1)^{1-\alpha}}{1-\alpha}+\frac{(k-1)\,(k-1)^{-\alpha}}{\alpha}\right] \\
&= h^{-\alpha}\left[\frac{k^{1-\alpha}}{1-\alpha}+\frac{k^{1-\alpha}}{\alpha}-\frac{k^{-\alpha}}{\alpha}-\frac{(k-1)^{1-\alpha}}{1-\alpha}+\frac{(k-1)^{1-\alpha}}{\alpha}\right] \\
&= h^{-\alpha}\left[\left(\frac{\alpha k^{1-\alpha}+(1-\alpha)\,k^{1-\alpha}-(1-\alpha)\,k^{-\alpha}}{\alpha\,(1-\alpha)}\right)-\left(\frac{\alpha\,(k-1)^{1-\alpha}-(1-\alpha)\,(k-1)^{1-\alpha}}{\alpha\,(1-\alpha)}\right)\right] \\
&= h^{-\alpha}\left(\frac{\alpha k^{1-\alpha}+k^{1-\alpha}-\alpha k^{1-\alpha}-k^{-\alpha}+\alpha k^{-\alpha}-\alpha\,(k-1)^{1-\alpha}-(k-1)^{1-\alpha}+\alpha\,(k-1)^{1-\alpha}}{\alpha\,(1-\alpha)}\right.
\end{aligned}
$$

Rearranging provides the weights for when $k = j$:

$$\alpha\,(1-\alpha)\,h^{\alpha}I_{intb} = (\alpha-1)\,k^{-\alpha}-(k-1)^{1-\alpha}+k^{1-\alpha}. \tag{3.105}$$

Interpolating Function near Singularity

Turning to the integral equation 3.100 with interpolating function 3.55:

$$
\begin{aligned}
I_{int\epsilon} &= \int_{(k-1)h}^{kh-\epsilon}\frac{u-(k-1)\,h}{h\,(kh-u)^{1+\alpha}}du \\
&= \frac{1}{h}\int_{(k-1)h}^{kh-\epsilon}\frac{u-(k-1)\,h}{(kh-u)^{1+\alpha}}du.
\end{aligned}
$$

Applying the substitution $u = (k-s)\,h \Rightarrow du = -hds$

$$I_{int\epsilon} = \frac{1}{h}\int_{u((k-1)h)}^{kh-\epsilon} \frac{(k-s)\,h-(k-1)\,h}{(kh-(k-s)\,h)^{1+\alpha}}\,(-h)\,ds.$$

Changing the limits of integration:

$$\begin{aligned}
I_{int\epsilon} &= \frac{1}{h}\int_{1}^{\frac{\epsilon}{h}} \frac{(k-s)\,h-(k-1)\,h}{(kh-(k-s)\,h)^{1+\alpha}}\,(-h)\,ds \\
&= \int_{\frac{\epsilon}{h}}^{1} \frac{(k-s)\,h-(k-1)\,h}{(kh-kh+sh)^{1+\alpha}}ds \\
&= \int_{\frac{\epsilon}{h}}^{1} \frac{kh-sh-kh+h}{h.h^{\alpha}.s^{1+\alpha}}ds \\
&= h^{-\alpha}\int_{\frac{\epsilon}{h}}^{1} \frac{1-s}{s^{1+\alpha}}ds \\
&= h^{-\alpha}\int_{\frac{\epsilon}{h}}^{1} \frac{1}{s^{1+\alpha}}-\frac{s}{s^{1+\alpha}}ds \\
&= h^{-\alpha}\int_{\frac{\epsilon}{h}}^{1} s^{-1-\alpha}-s^{-\alpha}ds \\
&= h^{-\alpha}\left[\frac{s^{-\alpha}}{-\alpha}-\frac{s^{-\alpha+1}}{-\alpha+1}\right]_{\frac{\epsilon}{h}}^{1} \\
&= h^{-\alpha}\Big(\left[\frac{1^{-\alpha}}{-\alpha}-\frac{1^{1-\alpha}}{1-\alpha}\right]-\left[\frac{\left(\frac{\epsilon}{h}\right)^{-\alpha}}{-\alpha}-\frac{\left(\frac{\epsilon}{h}\right)^{1-\alpha}}{1-\alpha}\right]\Big) \\
&= h^{-\alpha}\left(\frac{-1}{\alpha}-\frac{1}{1-\alpha}+\frac{\left(\frac{\epsilon}{h}\right)^{-\alpha}}{\alpha}-\frac{\left(\frac{\epsilon}{h}\right)^{1-\alpha}}{1-\alpha}\right) \\
&= h^{-\alpha}\left(\frac{-(1-\alpha)-\alpha}{\alpha\,(1-\alpha)}+\frac{(1-\alpha)\left(\frac{\epsilon}{h}\right)^{-\alpha}-\alpha\left(\frac{\epsilon}{h}\right)^{1-\alpha}}{\alpha\,(1-\alpha)}\right) \\
&= h^{-\alpha}\left(\frac{(-1+\alpha-\alpha)}{\alpha\,(1-\alpha)}+\frac{\left(\frac{\epsilon}{h}\right)^{-\alpha}-\alpha\left(\frac{\epsilon}{h}\right)^{-\alpha}-\alpha\left(\frac{\epsilon}{h}\right)^{1-\alpha}}{\alpha\,(1-\alpha)}\right).
\end{aligned}$$

Rearranging and considering the equation as $\epsilon \to 0$ provides the weights when $k = 0$

$$\alpha\,(1-\alpha)\,h^{\alpha} = -1. \tag{3.106}$$

Following the integration of the interpolating functions, the following equations for the weights θ_{jk} have been obtained:

$$\alpha\,(1-\alpha)\,k^{-\alpha}\theta_{jk} = \begin{cases} -1 & \text{when j = 0} \\ 2j^{1-\alpha}-(j-1)^{1-\alpha}-(j+1)^{1-\alpha} & \text{when j=1,2,3,...,k-1} \\ (\alpha-1)\,j^{-\alpha}-(j-1)^{1-\alpha}+j^{1-\alpha} & \text{when j=k} \end{cases} \tag{3.107}$$

Construction of algorithm

Returning to equation 3.98, for a given number of steps (n) an equispaced grid is formed where $t_k = \frac{k}{n}$ and $k = 1, 2, \ldots, n$. Applying the definition 3.99 to the problem:

$$f(t_k) + \beta y(t_k) = \frac{1}{\Gamma(-\alpha)} \int_0^{t_k} \frac{y(u) - y(0)}{(t_k - u)^{1+\alpha}} du.$$

Using the substitution $u = t_k - t_k w$ and resultant first derivative $\frac{-1}{t_k} du = dw \rightarrow du = -t_k dw$:

$$\begin{aligned} f(t_k) + \beta y(t_k) &= \frac{1}{\Gamma(-\alpha)} \int_{w(0)}^{w(t_k)} \frac{y(t_k - t_k w) - y(0)}{(t_k w)^{1+\alpha}} (-t_k)\, dw \\ &= \frac{-1}{t_k^{\alpha} \Gamma(-\alpha)} \int_{w(0)}^{w(t_k)} \frac{y(t_k - t_k w) - y(0)}{w^{1+q}} dw. \end{aligned}$$

Changing limits of integration:

$$\begin{aligned} f(t_k) + \beta y(t_k) &= \frac{t_k^{-\alpha}}{\Gamma(-\alpha)} \int_0^1 \frac{y(t_k - t_k w) - y(0)}{w^{1+\alpha}} dw \\ &= \frac{t_k^{-\alpha}}{\Gamma(-\alpha)} \left(\int_0^1 \frac{y(t_k - t_k w)}{w^{1+\alpha}} dw - \int_0^1 \frac{y(0)}{w^{1+\alpha}} dw \right) \\ &= \frac{t_k^{-\alpha}}{\Gamma(-\alpha)} \left(\int_0^1 \frac{y(t_k - t_k w)}{w^{1+\alpha}} dw - \left[\frac{w^{-\alpha} y(0)}{-\alpha} \right]_0^1 \right) \\ &= \frac{t_k^{-\alpha}}{\Gamma(-\alpha)} \left(\int_0^1 \frac{y(t_k - t_k w)}{w^{1+\alpha}} dw - \left[\frac{y(0)}{-\alpha} \right] + \left[0 \right] \right) \end{aligned}$$

$$f(t_k) + \beta y(t_k) = \frac{t_k^{-\alpha}}{\Gamma(-\alpha)} \left(\int_0^1 \frac{y(t_k - t_k w)}{w^{1+\alpha}} dw + \frac{y(0)}{\alpha} \right). \tag{3.108}$$

Returning to the definition of the Trapezium Rule 3.50, replacing the integral in 3.108 and remembering that $t_k = kh$:

$$f(kh) + \beta y(kh) = \frac{t_k^{-\alpha}}{\Gamma(-\alpha)} \left(\sum_{j=0}^{k} \theta_{jk} y(kh - khw) + \frac{y(0)}{\alpha} \right).$$

Replacing $w = \frac{j}{k}$ and using $y(\xi) \approx y_\xi$ where ξ is a dummy variable (Note: $f(kh) \approx f_k$).

$$\begin{aligned} f_k + \beta y_k &= \frac{(kh)^{-\alpha}}{\Gamma(-\alpha)} \left(\sum_{j=0}^{k} \theta_{jk} y(kh - jh) + \frac{y(0)}{\alpha} \right) \\ &= \frac{(kh)^{-\alpha}}{\Gamma(-\alpha)} \left(\sum_{j=0}^{k} \theta_{jk} y_{k-j} + \frac{y(0)}{\alpha} \right). \end{aligned}$$

Rearranging to make y_k the subject and utilising $h = \frac{1}{n}$:

$$\begin{aligned} \Gamma(-\alpha)\beta y_k &= \left(\sum_{j=0}^{k} \theta_{jk} y_{k-j} + \frac{y(0)}{\alpha} \right) - \Gamma(-\alpha) f_k \\ \left(\frac{k}{n}\right)^{\alpha} \Gamma(-\alpha)\beta y_k &= \sum_{j=0}^{k} \theta_{jk} y_{k-j} + \frac{y(0)}{\alpha} - \left(\frac{k}{n}\right)^{\alpha} \Gamma(-\alpha) f_k \\ \left(\frac{k}{n}\right)^{\alpha} \Gamma(-\alpha)\beta y_k &= \theta_{ok} y_k + \sum_{j=1}^{k} \theta_{jk} y_{k-j} + \frac{y(0)}{\alpha} - \left(\frac{k}{n}\right)^{\alpha} \Gamma(-\alpha) f_k \\ \theta_{ok} y_k - \left(\frac{k}{n}\right)^{\alpha} \Gamma(-\alpha)\beta y_k &= \left(\frac{k}{n}\right)^{\alpha} \Gamma(-\alpha) f_k - \sum_{j=1}^{k} \theta_{jk} y_{k-j} - \frac{y(0)}{\alpha} \\ y_k \left(\theta_{ok} - \left(\frac{k}{n}\right)^{\alpha} \Gamma(-\alpha)\beta \right) &= \left(\frac{k}{n}\right)^{\alpha} \Gamma(-\alpha) f_k - \sum_{j=1}^{k} \theta_{jk} y_{k-j} - \frac{y(0)}{\alpha} \end{aligned} \tag{3.109}$$

$$y_k = \frac{1}{\left(\theta_{ok} - \left(\frac{k}{n}\right)^{\alpha} \Gamma(-\alpha)\beta \right)} \left(\left(\frac{k}{n}\right)^{\alpha} \Gamma(-\alpha) f_k - \sum_{j=1}^{k} \theta_{jk} y_{k-j} - \frac{y(0)}{\alpha} \right). \tag{3.110}$$

As previously derived, the weights θ_{jk} are obtained from equations 3.107.

Diethelm-Chern Algorithm: Example

Let us consider an example using the Diethelm-Chern Agorithm. The example equation considered is given in Equation 3.111 as:

$$D^{1/2}[y - y_0](t) = -y(t) \text{ where } y_0 = 1 \tag{3.111}$$

Considering the step size $h = 0.1$ and the termination point of 1. From Equation 3.107 the weights of the algorithm are calculated and provided in Table 3.4. The solution at each step is provided in Table 3.5.

j	**k**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	-4	-5.657	-6.928	-8	-8.944	-9.798	-10.583	-11.314	-12.048	-12.649
1	2	3.311	4.068	4.686	5.230	5.743	6.234	6.659	7.058	7.415
2		0.339	0.666	0.768	0.857	0.941	1.021	1.091	1.156	1.088
3			0.204	0.4	0.446	0.490	0.532	0.568	0.600	0.630
4				0.144	0.285	0.313	0.340	0.364	0.385	0.404
5					0.108	0.221	0.239	0.261	0.271	0.285
6						0.088	0.181	0.193	0.208	0.218
7							0.082	0.170	0.176	0.184
8								0.057	0.136	0.142
9									0.055	0.123
10										0.054

Tab. 3.4: Weights for Diethelm-Chern example

k	y_k
0 (initial condition)	1
1	0.781
2	0.680
3	0.619
4	0.574
5	0.539
6	0.511
7	0.490
8	0.471
9	0.452
10	0.433

Tab. 3.5: Value of y_k

3.5 Challenges with numerical method implementation

Numerical methods can be challenging to implement depending upon the method chosen. In [15] the Authors set out areas which they considered in assessing numerical schemes for fractional differential equations. These areas were based upon the established 'standards' for numerical solutions to ordinary differential equation and were: *convergence*, *consistency*, and *stablity* (see Section 3.2.1). Diethelm et al [15] additionally considered requirements which would enable efficient implementation of the numerical method into a computer program. These requirements were: how easy the method was to program into a suitable computer set-up, how reliable the results were from said program, and how swift the program was to run. The consideration of how easy a suitable computer set-up was to program and the speed that computation completed are key features of this research and will be discussed further in the subsequent chapters.

3.5.1 Improvement of 'accuracy' including Richardson Extrapolation

The order of convergence indicates the speed with which a numerical method converges to the exact solution as the step size decreases. A smaller step size will increase the number of steps required to complete the period of integration and will consequently increase the number of calculations. Although the accuracy will have been enhanced, the increased calculations will mean increased computational effort, slowing the method's conclusion. A balance therefore has to be achieved in order to gain a relatively 'accurate' solution to the differential equation without decreasing the speed with which the calculations can be completed.

The application of an extrapolation scheme can assist in increasing the accuracy of a numerical method whilst maintaining a low number of steps. Extrapolation will increase some computational effort but this should be minimal compared to the alternative of successively smaller step sizes to achieve the desired level of accuracy. One such extrapolation technique is 'Richardson Extrapolation' also referred to as "Deferred approach to limit" ([40]).

In Lambert [40] and the paper by Faragó et al [24] Richardson Extrapolation is derived for ODEs. To summarise, consider the exact solution $y(t)$ at the point $t_n \in [a, T]$. Two approximations (z_n and w_n) are derived through the use of the same convergent numerical method of **order p**. The first of these approximations calculates the solution using step size h, the other with

step size 0.5h. This results in the following formula:

$$y(t_n) = z_n + h^p Q + O\left(h^{p+1}\right) \tag{3.112}$$
$$y(t_n) = w_n + (0.5h)^p Q + O\left(h^{p+1}\right). \tag{3.113}$$

Q is dependent upon the convergent numerical method used and consists of elementary differentials. Ignoring the $O(h^{p+1})$ terms and multiplying Equation 3.113 by 2^p yields:

$$\begin{aligned} 2^p y(t_n) &= 2^p w_n + \frac{2^p}{2^p} h^p Q \\ 2^p y(t_n) &= 2^p w_n + h^p Q. \end{aligned} \tag{3.114}$$

Subtracting 3.112 from 3.114 enables the elimination of the 'Q' terms:

$$\begin{aligned} 2^p y(t_n) - y(t_n) &= 2^p w_n - z_n \\ y(t_n)\left(2^p - 1\right) &= 2^p w_n - z_n \\ y(t_n) &= \frac{2^p w_n - z_n}{2^p - 1}. \end{aligned} \tag{3.115}$$

The order of equation 3.115 is $O(h^{p+1})$ and indicates the result will be of greater accuracy than the numerical method alone ([24] and subsection on 'Convergence'). However the accuracy of the extrapolated numerical method is dependent upon the selection of a suitably small step size.

The use of Richardson Extrapolation to control the step size by a predetermined level of accuracy is one form of implementation. Faragó et al [24] additionally describe two other ways to implement Richardson Extrapolation: Active, and Passive. 'Active Richardson Extrapolation' utilises the "hopefully improved approximation" to $y(t_n)$ in successive calculations. Theoretically this 'active' implementation of Richardson Extrapolation should yield improved results however Faragó et al found this is not always the case. In particular unstable results were yielded when the Active Richardson Extrapolation was applied with the Trapezium Rule. Alternatively 'Passive Richardson Extrapolation' does not utilise the 'improved' approximation for $y(t_n)$ within the procedure, rather uses the improved calculations after the fact for an improved final step. Due to the potential application of Richardson Extrapolation with methods based upon the Trapezium Rule, the passive form was explored further within this thesis (Chapter 7).

3.6 Discussion Points

As a result of this Chapter, background information on numerical methods and challenges faced in their application have been discussed. Some general methods for ordinary and fractional differential equations were explored. The Richardson Extrapolation Scheme was presented as a tool in increasing accuracy of numerical method results.

The information presented in this Chapter has provided the following discussion points which will be taken forward in this research:

- Basic understanding of what a numerical method is and how to assess for an appropriate scheme.
- Some general methods for ordinary differential equations, in particular the Runge-Kutta Method.
- Some general methods for fractional differential equations (Lubich's Fractional Multistep Method, Fractional Adams Method and Diethelm-Chern Method)
- Importance of step size and how accuracy is increased through the application of Richardson Extrapolation without the need to decrease the step size.

4. PARALLEL COMPUTING

4.1 Objectives

This Chapter will provide an understanding of what is meant by parallel computing. In particular, details of what architecture and software are needed to create a parallel environment will be discussed. The concept of multithreading will be defined. The Chapter will conclude with techniques employed by programmers to assess the effectiveness of created parallel programs.

To summarise this Chapter will satisfy the following objectives:

- To provide an understanding of what parallel computing is.
- To outline how parallel computing can help in the solution of differential equations.
- To provide a brief on software and hardware needed to achieve parallelism.
- To define 'multithreading'.
- To present the challenges faced when implementing an algorithm in parallel.
- To outline a way to assess efficiency of parallel programs.

4.2 Introduction to Parallel Computing

What is parallel computing? Parallel computing is "a collection of processing elements that can communicate and co-operate to solve large problems fast" [2]. By processing elements, this definition is referring to a group of processors, or elements of a processor, working together to solve a problem. However, in order to make such an architecture work, parallel computing also encompasses the ability to divide a program, series of tasks, or data set into groups which can be operated simultaneously. For example, two processors

operating the same program using their own data set. Such a situation might occur when modelling the impact of extreme weather on different countries.

Parallel computing has been around for decades. The First Supercomputer was thought to have been built for NASA in 1967 [2]. This form of parallel architecture represents a traditional image which still remains current with the top performing computers in the world [64]. Supercomputers have the *Capability* to complete very large complex tasks where speed and availability of resources (e.g. memory) is of the essence. Such large complex tasks would not have been possible without an integrated, and capable, parallel architecture. However the cost to purchase and maintain such an architecture means that preparatory work is advisable. Thankfully the resources made readily available through desktop machines, operating in clusters, can provide the preparatory solutions for Supercomputers [63], see Section 4.4 for more information.

4.2.1 Flynn's Taxonomy

Flynn's Taxonomy (circa. 1966/1972) describes the basic types of computing architectures [10] [59] [6] which are:

- SISD - **S**ingle **I**nstruction acting upon **S**ingle **D**ata. As the name suggests a single instruction is processed sequentially upon a single data stream. Doubling the speed of the processor will halve the time taken to complete the computation. An example of an SISD structure is the von Neumann architecture [59]. This form of architecture is not considered to be parallel.

- SIMD - **S**ingle **I**nstruction acting upon **M**ultiple **D**ata. An example of SIMD is a vector processor where the hardware and software is segmented akin to a factory production line. The elements of the production line take segments of the computation which remain mainly independent of each other. The data proceeds down the production line called the 'pipeline'. An application for this type of architecture is image processing [6]. This aspect can be seen as a form of parallel computing.

- MISD - **M**ultiple **I**nstructions acting upon a **S**ingle **D**ata source. This type of computing scenario is very rare but might occur in e.g. code breaking where mutliple processors attempt different techniques upon a single coded message. This is also seen as a form of parallel computing.

- MIMD - **M**ultiple **I**nstructions acting upon **M**ultiple **D**ata. This aspect of the Taxonomy is seen as true parallelism. There are many examples of MIMD from: the multicore processing power in standard desktops, to clusters of computers working in parallel (e.g. Beowulf Clusters, see Section 4.4.1), to the supercomputers found in the Top 500 best performing computers in the world [64].

The **S**ingle **P**rogram **M**ultiple **D**ata form of parallel computing can be seen as a subset of MIMD. SPMD allows the programmer to send out a single program which operates on all processors using their own data. For example: suppose there is a pair of computers working on a program to calculate the exponential of the elements in an array 'X'. Each processing entity takes half of the array 'X' and calculates the exponential, writing the results back to a new array 'Y'. (This example is illustrated diagramatically in Figure 4.1.) MatLab Parallel Computing Toolbox (see Chapter 5) provides a function 'SPMD' which facilitates this type of parallel programming. The concept of SPMD will be taken forward within this research. For example: To allow separate processing units to operate an algorithm with different step sizes, the results from each processing unit will be communicated back to a central source to allow Richardson Extrapolation (see Chapter 7).

Parallel programs can additionally be defined within a continum of granularity. The coarser the granularity in the program, the increased amount of work which can be completed in parallel without the need for synchronisation to complete the program. From Shonkwiler et al [59] the key points in the continum are:

- Fine grained - small subtasks of the program are completed in parallel. For example consider the calculation of elements of a matrix by means of a for-loop[1]. Each iteration of the loop is divided amongst the processing entities. So that, for example, the first n entries are calculated by processor 1, the second n entries by processor 2 and so on.

- Coarse grained - Major tasks in the program are completed in parallel. For example consider a group of processing entities, each entity has its own set of tasks to perform which are independent from the rest of the groups. An initial command provides a single set of data for each processing entity to use with its set of tasks. Once all tasks are complete the results for each processing entity are communicated across all workers in the group.

[1] By 'for-loop' we are referring to a series of tasks which are repeated multiple times in order to complete a job.

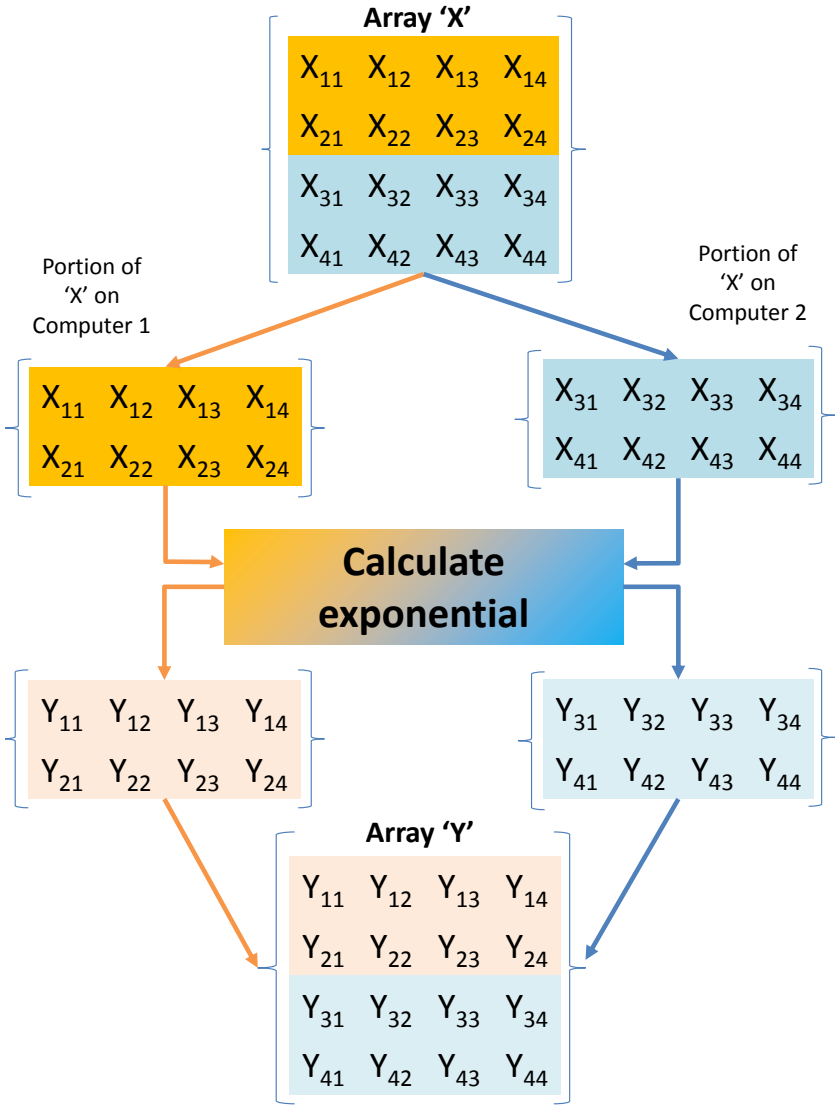

Fig. 4.1: Potential use of SPMD: Calculation of array 'Y'

- Very coarse grained - Very few (if any) points of communication between processors. Also referred to as 'Asynchronous parallelism' or 'Embarassingly parallel'. For example each processing entity has its own matrix to work upon. Unlike the coarse grained example, no synchronisation of the resultant data is undertaken.

The concept of granularity can also be applied to parallel architecture. Coarse granularity refers to relatively powerful computers working together in a cluster. Consequently fine granularity refers to low performing computers in a cluster, and very coarse granularity would refer to supercomputers or high performing computing architectures [10].

4.3 Why use parallel computers?

Barney [6] provides a list of reasons why parallel computers should be used, these are:

- To provide concurrency - In other words the ability to do many things at the same time.

- To solve large or complex problems - Some problems are difficult or impossible to complete sequentially or on a single computer. Large or complex problems include the 'Grand Challenge' Problems i.e. "a fundamental problem in science or engineering, with broad applications, whose solution would be enabled by the application of the high performance computing resources" [28]. Examples: modelling climate change, modelling the formation of galaxies.

- To save time and potentially money - Additional resources should enable the speedier solution of large or complex problems. If the time to solution is quicker than the sequential equivalent, it should be possible to save money through e.g. reduced energy consumption.

- To improve the use of multicore technology - Standard desktop computers have multiple processors or "cores" thereby creating their own parallel environment. The application of parallel computer programming to such an environment will better utilise this architecture.

- To use resources which are not local - This aspect will allow the programmer to use resources which may not be available in the local environment e.g. specialist software, increased memory capacity etc.

4.4 Capacity Parallelism

As discussed in Section 4.2 preparatory work for supercomputer implementation is advisable/needed, this is delivered through *Capacity computing*. Stohmaier et al [63] define Capacity computing as a "smaller or cheaper system...where smaller problems are solved". Capacity computing systems allow programmers to prototype alternatives to existing program designs, and to conduct parametric studies. The stablisation of communication and technology, coupled with better performance has resulted in an increased use of cluster systems (e.g. Beowulf Cluster, the Grid) to perform 'capable' work i.e. work which would have been undertaken by a supercomputer set-up.

4.4.1 Examples of Capacity Parallelism

Beowulf Cluster

A 'Beowulf Cluster' is a term used for a group of standard desktop computers connected through a dedicated local network, working in parallel to solve problems when not in traditional use. From Sterling [61], the main features of a Beowulf Cluster are:

- Computers in Clusters are commerically available - this allows for the easy replacement of components which are not operational or allows new technologies to be integrated into the Cluster. These computers are less costly than an alternative supercomputer system.

- Software is of no/low cost - Middleware used to control each computer and the connection between the computers in the Cluster attempts to be platform independent i.e. independent of operating system such as Windows, Linux etc.

- Price vs performance ratio - Due to the low cost of purchasing a Cluster and software, the performance achieved can be very satisfactory when comparing to alternative supercomputer set-up.

- Flexibility in configuration - The Cluster can be configured by local administrators. This allows additional flexibility in the configuration of the Cluster e.g. using particular processing entities for tasks which they are better suited. For example: increased memory on a particular element of the Cluster could enable the manipulation of large matrices.

'The Grid'

'The Grid' vision is a global cluster where thousands of computing resources would be linked together across the web to form a massive, super-powerful, computer [29]. By computing resources we are refering to: computers, laptops, data archives, telescopes, sensors etc.

Although 'The Grid' vision has not reached fruition, smaller grids have been around for some time in the scientific community e.g. the seti@home project [56]. But what makes a grid different from any other cluster? Essentially it is the use of the web to join computing resources together which has changed how users of grid technology interact with each other and how they operate the resources. To ellaborate, Dongarra [21] indicates that grid technology has moved the traditional methods used in parallel computer programming ('batching' jobs off to elements of the cluster) to incorporate increased interactivity with resources. This has been made possible through the increased capabilities in network speed which doubles in performance every nine months compared to every eighteen months for processing power [26]. In addition, the use of resources beyond a private network mean protocols and interfaces have to be developed to allow the user to access resources securely, quickly, and without too much fuss. By using the web, users of a grid also have the opportunity to interact with each other, collaborating on problem formation, analysis etc.

Foster [26] indicates the layers which a grid requires, these are:

1. Fabric - networked resources such as computers, laptops, data storage, sensors, telescopes etc which will be shared/accessed.

2. Resource and connectivity protocols - core communication and authentication protocols. This layer allows exchanges of data between resources using secure mechanisms to verify identity and access. The initiation, monitoring and control of the resource sharing operations are also included at the Resource Layer.

3. Collective services - this layer enables interactivity across collections of resources e.g. brokering, directory services, monitoring and diagnostic services, services to identify who can access the resources (membership).

4. User applications - this layer enables the user to call elements of the other layers.

In the article by Foster [26] grid technology has to address the following areas before implementation:

1. Knowing what network/resource are available - how will the user discover the resources exist?

2. Obtaining permission to use network/resource - assuming a user knows a resource exists, permission will need to be gained to access the resource. Will there be conditions in the resource's use?

3. Cost of using network/resource - will remuneration be required to use the resources?

4. Accessibility of the network or resources i.e. software - getting the user's resources to work with the rest of the grid.

5. Security of own network - a user is opening a connection to an unknown resource. How secure and virus-free is the resource?

6. Security of network/resource which is being accessed - same as previous point but from the perspective of the entity and its owner.

7. Capability of accessed network/resource - how good are the resources being accessed? Will they be capable to complete the program?

Brooke and Parkin [8] indicate that the software or 'middleware' which is used to facilitate the operation of grid technology causes issues for short term implementation of a grid. In particular they note that:

- The most popular software or middleware used to support grid technology is Globus. The installation of Globus is resource intensive for the user's computational resource.

- From both a user and supplying resource perspective, the middleware needs to have customised libraries which help to secure the grid resources. These libraries need to be altered whenever the operating system sends out a security patch. In particular Brooke and Parking explain that most users of a grid may be detered in using the technology as they need to use and maintain digital certificates which is "...personal data that can be used to identify the holder" [1].

Brooke and Parkin go on to suggest short term grids could be formed as an 'Alliance', where coupling between the user and the resource provider is looser than the long-term counterpart. To facilitate the Alliance, Brooke

and Parkin recommend access to resources is defined based upon Role Based Access Control (RBAC) where permissions are defined for groups of users in particular roles of an organisation. They also suggest a 'Grid Ontology' to remove the diversity which now occurs in the middleware. This ontology would be defined upon common processes in grid technology, with bolt on components reflecting specific grid middleware or the structure of the user's organisation. The looser relationship and identified suggestions would enable more dynamism and lighter implementation, resolving the issues Brooke and Parkin identified. A lighter implementation could also facilitate the inclusion of 'lighter' devices such as mobiles or tablets to be part of a grid architecture.

Despite the limitations to grid technology the potential for increased resources through the web offers greater opportunities for parallel computing which have only become more readily available in recent years. Since the publication of the article by Foster [26] elements of grid technology are already present in everyday life. Examples include: Cloud technologies to store data, software etc; crowdsourcing for collaborative effort such as the Planet Hunters Project.

Multicore and Multithreading

Standard desktop computers now contain an increased number of processing elements or **cores** in order to increase performance [36].

Additionally **multithreading** allows increased computational power within a processing core. A 'thread' is a "line or flow of control through a program" [59]. By 'multithreading' we are referring to the capability to run multiple threads concurrently of each other [12]. An example of multithreading could be one thread for opening a word processing document whilst a second thread could check for new email in another program.

Derived from multithreading, many Intel Processors (see Chapter 6) implement hyperthreading technology. Hyperthreading Technology refers to hardware which, when implemented, allows the creation of two 'logical' processors per core. Within the core, the system divides, copies or shares resources to allow the logical processors to operate two threads concurrently [35]. When a logical processor has completed its thread, the resources are released allowing the other logical processor to complete. (Hyperthreading was briefly abandoned by Intel when multicore was introduced. Hyperthreading was reintroduced in 2008 see [34]).

Techniques from the parallel computing community can be transferable to common computing practice given the multicore and multithreading technology advances [11]. Irrespective of this potential for increased popularity, multicore technology can be used to prototype programs for larger multiprocessor, multicore architectures. This research has developed prototype parallel programs for a multi-core processor architecture (see Chapter 6) which resulted in the availability of eight processing entities/workers for parallel processing purposes. This eight worker environment could be adapted to much larger computing architectures, dependent upon how easily the problem lends itself to an increased number of processing entities. For example: if the underlying data for processing was very large, benefit could be gained. However an increased number of processing entities/resources could increase communication overheads between entities and therefore a balance needs to be achieved.

The MatLab Parallel Computing Toolbox environment enables implicit multithreading with no user input. This implicit multithreading is discussed further in Chapter 5 and 6.

4.5 Implementation of parallel programs

In Dongarra et al [21] consideration is given as to how programs can be structured in parallel form. This includes the method that parallelism can take i.e. task and/or data parallelism. By 'task parallelism' we are referring to processing entities running major elements of the program in parallel on the same data. For example: the application of different real-world models based upon the same data such as the assessment of risk in infection control. By 'data parallelism' we are referring to the division of the data into smaller components to be processed by individual entities using the same task(s). For example: consider the multiplication of matrices A and B. In this example each processor would take a proportion of matrix A and multiply it to the replicated matrix B. In this scenario communication between workers would be required to assist in completing the set task.

Additionally, Dongarra [21] references two implementation styles for parallel programs:

- Parallel Loop - in this style portions of the loop are assigned to different processing entities. In traditional parallel programming the assignment of loop iterations is specified by the programmer. However, in MatLab

Parallel Computing Toolbox (MPCT) the use of the high-level construct 'parfor' means the assignment of loop iterations is undertaken autonomously by the software (see Section 5.3).

- SPMD - as described in Section 4.2.1 each processing entity applies the same code to different data sets. Again MPCT provides a high-level construct 'spmd' which facilitates this style (see Section 5.3).

An important tool which is pursued in this research is the concept of a 'block' [14]. Consider a data set divided into groups determined by the number of processing entities within a cluster. Each group is referred to as a 'block'. As each block is processed, every computing entity operates on its own data allocated within the block. As the parallel program continues, communication between workers occurs so that each computing entity can complete its tasks, and the group can complete the current block. Once a block is complete communcation between workers occurs again to share results. The parallel program moves the cluster to the next block for processing. The cycle continues until all blocks, and hence data, have been processed. This style of programming allows processing entities to calculate independent elements of the program whilst awaiting fresh data from others in the cluster. The block method is used in this research to divide the total number of steps used to approximate a differential equation using a numerical method, see Chapters 8 - 10.

4.5.1 Software for parallel programming

Common programming languages such as FORTRAN and C++ are used to write programs for the parallel environment. A separate message passing interface (such as MPI) will be required to communicate between processing entities in a distributed memory environment. The MatLab Parallel Computing Toolbox (previously called the Distributed Computing Toolbox) was launched in 2004 following previous user-developed incarnations such as pmatlab. MPCT allows the user to construct programs which take advantage of the existing accessible MatLab language and build parallel programs using high level constructs. Parallel Computing Toolbox combines both programming and communciation elements together and has been investigated within this research see Chapter 5.

4.6 Evaluation of Effectiveness

In order to evaluate effectiveness we have to ask ourselves some key questions:

- Has the solution to the problem been found using the parallel program? Do the results match that of the sequential program equivalent?

- What is the speed up? In other words the time it takes for the parallel program to finish in comparison to the sequential equivalent.

- Is the speed up scalable or close to scalability? In other words, if a program operating on a single worker executes in a time $T(s)$ then does the same program operating on a cluster of x workers execute in a time close to $\frac{T(s)}{x}$.

- Is the distribution of load appropriate between processing entities?

- What runtime is predictable from the floating point operations and how does this compare to the real runtime? (See Section 4.6.3)

- Can we measure feasibility? In other words the cost of creating the parallel program compared to the man-hours saved through the creating and use of the parallel program.

- What is the theoretical benefit of creating the programs in parallel compared to sequential implementation i.e. reduction in number of calculations? How does this compare to the practical experience of running the parallel programs and their sequential equivalent?

4.6.1 Speed up

In Dongarra et al [21] speed up equations are provided. Two of which are:

$$\text{Speedup } (n) = \frac{T(1)}{T(n)} \tag{4.1}$$

$$\text{Speedup } (n) = \frac{T_s + T_p}{T_s + \frac{T_p}{n}} \leq \frac{T(1)}{T_s} \ldots \text{Amdahl's Law} \tag{4.2}$$

$$\tag{4.3}$$

In equation 4.1 $T(1)$ refers to the time for the sequential program to complete, $T(n)$ is the time for the parallel program to complete across its n processing entities. The point at which parallel implementation takes the same or less time than a sequential equivalent has been referred to in this research as the 'Tipping Point' (see below)

Equation 4.2 adapts equation 4.1. Equation 4.2 additionally considers the tasks in the program which have had to be completed sequentially and those which are undertaken in parallel. In the formula T_s is the time required for the sequential portion of the program, T_p is the time for the parallel portion of the program, and n is the number of processing entities. Equation 4.2 is called Amdahl's Law which states that speed-up is relative to the amount of the program which has to be run sequentially. As stated in [21] if 20% of the program has to be computed sequentially then we cannot expect the speed-up to be greater than 5.

The choice of speed up equation should be determined by the form of parallel program implemented. If a program has a large amount of sequential computation Equation 4.2 should be used. However, for the purposes of this research, the Speed-up Equation 4.1 will be taken forward as a tool for analysing execution time of parallel code.

'Tipping Point'

As indicated in the previous section, the 'Tipping Point' refers to the point at which a parallel program takes the same or less time than a sequential equivalent. In order to discover the 'Tipping Point' it becomes necessary to increase the size or complexity of the problem which is being solved. If the problem is too small or simple the communication overheads incurred through implementing the problem in parallel would be too significant. The runtime of the program would be taken up with sharing or passing data between workers rather than calculating results.

In this thesis we will be considering numerical methods for solving Fractional Differential Equations which are solved over an increasing number of steps. The discovery of the 'Tipping Point' provides a superficial indication of a point where the numerical method may benefit from parallel implementation.

4.6.2 Theorectial versus practical benefit of implementing numerical method in parallel

Theoretically it could be assumed that running a program across four processors would take a quarter of the time to execute compared to running the program sequentially on one processor. As alluded to earlier, this theory called 'Scalability', which will be discussed in Section 4.7.1, does not occur

practically due to other overheads such as communication. Minimising the communication between workers by decreasing the granularity of a program and/or packaging more than one piece of data to another worker(s) can reduce the overheads and help increase efficiency of parallel programs.

Although the aggregation of execution data will enable a calculation of the practical benefit of 'parallelising' a program, the theoretical benefit which could have occurred provides a useful analysis tool. The theoretical benefit of implementing a numerical algorithm in parallel versus a sequential equivalent can be calculated through the total number of ***actions*** required to run the program. An 'action' refers to the retrieval and storage of data, or calculations such as addition and multiplication. By comparing the number of actions required to complete a parallel program in comparison to its sequential equivalent, a percentage increase/decrease can be calculated. Comparing the theoretical increase/decrease of actions against a similar percentage using aggregated execution time data (practical benefit calculation) it will be possible to identify parallel programs with a poor load-balance (See Section 4.7.2). The calculation of the number of actions within each line of code also enables the identification of the most computationally expensive operations which is useful for calculating the floating point operations (see Section 4.6.3)

Let us look at an example which illustrates how the 'number of actions' has been calculated. In a program a line of code reads,

```
a = b + (4 * c);
```

this can be interpreted as,

```
fetch c
multiply by 4
fetch b
add b + (4*c)
store as variable a
```

This one line of code would therefore become five actions.

4.6.3 Floating Point Operations (flop)

Golub et al [27] define a 'flop' as "a floating point add, subtract, multiply or divide." The calculation of the 'number of flops' provides a function, for which the dominant term determines the most computationally expensive aspect of an algorithm or program. For example:

```
1 for i = 1:r
2     x = a+b;
3     for j = 1:i
4         y = x * a;
5     end
6 end
```

The most expensive computational aspect of the program above is in line 4 where the line is part of a nested for-loop. Line 4 contains one multiplication operation, however this is run an increasing number of times depending upon the value of i within the outer loop. The number of flops is therefore:

$$\begin{aligned}\sum_{i=1}^{r} i &= 1+2+3+\cdots+r \\ &= \frac{r\,(r+1)}{2} \\ &= \frac{r^2}{2}+\frac{r}{2} \end{aligned} \tag{4.4}$$

The dominant term in Equation 4.4 is r^2 and therefore the order of arithmetic is $O\,(r^2)$ with the order constant $\frac{1}{2}$.

The division of the dominant term by the operations per second of a computer can determine the lowest bound for the program runtime [62]. However, this **flop theoretical runtime** will not be translated to the real runtime of the program which will be impacted by other overheads such as communication. Stewart [62] indicates that proportionality exists between the flop theoretical runtime and the real runtime of a program in a large number of cases.

The flop calculation is used to further analyse program performance in this thesis (see Chapters 7 to 10).

4.7 Challenges in implementing Parallel Computer Programs

4.7.1 Scalability

The speed with which the parallel computing environment completes the program is unlikely to achieve scalability when comparing to the sequential equivalent. By 'Scalability' we are referring to "a parallel system's

(hardware and/or software) ability to demonstrate a proprionate increase in parallel speedup with the addition of more processors" [6]. For example, you would assume that increasing the number of processing entities to four would result in $\frac{\text{sequential run time}}{4}$. The result in this example will rarely be the case. Why? Scalability depends upon the architecture and/or the program. Potential issues to achieving scalability:

- Performance of processing entities - The performance of processing entities may be problematic in a composite cluster as found in a Beowulf Cluster or a Grid. The time for a parallel program to complete will only be as quick as the slowest member.

- Connection issues - If in some form of network or internet cluster, the connection speed will impact upon performance and could be deteriorated further if the line is shared with other users i.e. traffic congestion.

- Synchronisation of processing entities - Communication points within a parallel program will hinder runtime as processing entities wait to synchronise. This is especially pertinent if there are performance issues with a processing entity or if the amount of work assigned to an entity is too lengthy for performance/balance (see Section 4.7.2). Reduction in synchronisation points can be achieved by aggregating data transfers although care has to be taken in ensuring that data is made available at the appropriate time for other processing entities.

4.7.2 Distribution of load

'Load Balancing' refers to the even distribution of tasks amongst processing entities of assumed equal capability [10]. Extending this definition further: 'load balancing' also assumes a fast, even connection between processing entities. Poor load balancing can result in delays whilst other processing entities have completed their tasks or are awaiting data from the active entity. However, the distribution of workload between entities becomes more complex when dealing with differences with processing performance, connection issues, and where a processing entity has particular hardware or software capabilities. Examples: where there are connection issues - a program of coarse granularity might be preferable to reduce communication, or where one entity has larger memory capacity than its peers it may be preferrable to give a large matrix to this entity for processing.

4.7.3 Cost

As indicated earlier, cost savings can be achieved. However the cost of using, buying, or hiring a supercomputer is large. The problem which is to be solved has to justify this cost. As indicated earlier, the gap in capability of the supercomputers and computer clusters is narrowing. Therefore considering a computer cluster to solve a large or complex problem could be a viable alternative at a reduced cost.

4.8 Discussion Points

In this Chapter background information has been provided to aid understanding of parallel computing. General information was provided regarding potential parallel architectures such as multiprocessor/multicore and cluster environment provided on local area networks (Beowulf Cluster) or over an internet connection ('The Grid'). Structures for creating parallelism within programs were also discussed, with the concepts parfor and spmd. This Chapter also gave the Reader some ideas as to challenges experienced with parallel computing and how effectiveness can be measured.

As a result of this Chapter the following discussion points will be taken forward:

- Information regarding parallel architectures.
- Benefits and challenges of implementing parallel computing.
- Parallel loops and the SPMD forms of implementing a parallel program.
- Methods of assessing 'efficiency' of a parallel program in comparison to sequential equivalent.

5. MATLAB PARALLEL COMPUTING TOOLBOX

5.1 Objectives

Following Chapter 4, this Chapter looks specifically at MatLab Parallel Computing Toolbox (MPCT) as a language for programming a parallel architecture. Information is provided to the Reader regarding the high-level constructs available when prototyping parallel programs. Additionally the commands used for communicating between workers are provided for use in later chapters. The MatLab Profiler is also discussed as a method for assessing the efficiency of the prototype programs.

To summarise, this Chapter aims to satisfy the following objectives:

- To provide an outline of MPCT.
- To give constructs which can be employed when programming in parallel using MPCT.
- To provide commands employed by MPCT to communicate between workers.
- Highlight tools available in MPCT which can assist in assessing the efficiency of a parallel program.

5.2 What is 'MatLab Parallel Computing Toolbox' (MPCT)?

Traditional MatLab is a high-level language which was initially designed for matrix computation. In Hunt et al [33] the Authors describe MatLab as having evolved to include:

- the graphical representation of functions.
- the solution of equations.
- the calculation of statistical tests.

- the creation of sound and animation.
- the simulation and modelling of situations (enhanced if SIMULINK Software is installed).
- preparation of materials for the web.
- enhanced documentation through collaboration with Microsoft Office Software.

In 2004 the MathWorks extended the existing MatLab software into the parallel programming environment with the creation of MatLab Parallel Computing Toolbox (then called Distributed Computing Toolbox). There had been previous incarnations of a parallel implementation of MatLab such as pMatLab and MatLabMPI (developed by the MIT Lincoln Laboratory) [58]. However these versions of 'parallel MatLab' were developed by the user community unlike the subject of this research, MPCT.

By extending the existing MatLab software, the MathWorks have developed an easy to use environment for prototyping parallel programs. No separate message passing software is required for communicating between workers or **'LABs'** in the pool. High-level constructs, such as parfor and SPMD (see Section 5.3), enable communication between workers without the user's knowledge.

Although MPCT can be used to create a local pool of workers for executing parallel programs, MatLab Distrubuted Computing Server (also launched in 2004) can be installed on a remote cluster which therefore allows batching[1] of parallel programs.

Initial implementation of MPCT aimed at very coarse or 'embarassingly parallel' programs i.e. where no/little communication existed between workers. Such a situation might occur when forecasting or assessing risk. As the development of MPCT has become more refined, more features such as parallel loops and message passing capabilities have been added [58]. Also the adoption of implicit multithreading for enhancement of parallel capabilities on single mulitcore machines has been integrated into MatLab since version 7.4 (R2007a) [48] [47]. This multithread implementation causes some problems when comparing performance of sequential to parallel programming techniques which have to be resolved (see Chapter 6).

[1] By 'batching' we are referring to a local 'client' computer instructing a head node within a remote cluster to undertake a program, job or series of commands.

5.3 Implementation of parallel programs in MPCT

Sharma et al [58] indicate the design goals of The MathWorks in the development of MPCT. These goals were:

- Execution of arbitrary MatLab code and SIMULINK models on a cluster.
- Allow users to utilise existing knowledge of MatLab code in creating/operating parallel MatLab programs.
- Remove the need for users to control the parallel environment with respect to: architecture, specification of the system, multithreading, data management and synchronisation.
- Allow programs to be independent of the size of the parallel architecture available and enable appropriate scale-up within the architecture available. Where no parallel environment is present, programs should still be able to operate.
- The ability to create, correct, maintain, execute programs is more important than other matters such as performance.

With these design goals in mind The MathWorks provide several high-level components to enable parallel program creation in MPCT [44]:

- Parallel for-loops
- Distributed arrays and Single Program Multiple Data
- Batch jobs

In addition to the components above, and in keeping with providing a familiar environment for existing MatLab users, MPCT provides an interactive programming mode called 'pmode'. The pmode environment allows dynamic prototyping with the LABs in the parallel architecture. In other words it allows users to test parallel programming constructs without the need for creating m-files. The dynamic environment facilitates the instant display of results as and when the LABs complete which can be particularly valuable when learning more about MPCT.

5.3.1 Parallel for-loops

A 'for-loop' allows the repeat operation of a set of programming commands for a predefined number of times. Parallel for-loops enable a user to divide the iterations of a 'for-loop' amongst available workers. For example: if an existing 'for-loop' operates 500 times, a parallel for-loop would divide these iterations across a pool of four workers so that each would conduct 125 iterations or loops. Although no communication occurs between the workers within the parallel for-loop, unseen communication overheads would occur in organising the workers in the pool to undertake the task. Therefore a parallel for-loop is valuable when there are many iterations in the loop, or when the time taken to complete one iteration is lengthy [44]. Alternatively, parallel for-loops can be used for task parallelism, where each iteration references a different set of tasks e.g. alternative weather models using the same initial or boundary conditions.

One way MPCT enables a parallel for-loop is through the 'parfor' command which replaces the existing 'for' command. A parfor-loop can be used where each loop iteration is independent of order and of each other [44]. For example, let us consider a parfor-loop as detailed below and given in [44]. This example demonstrates how parfor works and not necessarily the best application. The loop contains 10 iterations which need to be performed across a pool of 'u' workers. In this situation each worker would undertake $\frac{10}{u}$ iterations of the loop. The MPCT user is unaware of which LAB is undertaking which iterations, or the sequence with which the iterations are being operated. This would not matter in the example below.

```
x = 0;
parfor i = 1:10
    x = x + i;
end
x
```

Despite reference to the same variable 'x' the correct result would be obtained by the 'Client'[2] machine which would accumulate the final result.

To illustrate the case where a parfor-loop **would not operate** correctly, let us consider the following example. In this case each iteration of the loop means r is divided by a different element of the array t. If this order is

[2] By 'Client' we are referring to the computer which initiated the jobs/tasks [44]

```
>> ParforExample
With A MatLabPool in operation.  Parfor loop operational
Starting parallel pool (parpool) using the 'local' profile ... connected to 2 workers.

ans =

 Pool with properties:

            Connected: true
           NumWorkers: 2
              Cluster: local
       AttachedFiles: {}
         IdleTimeout: 30 minute(s) (30 minutes remaining)
         SpmdEnabled: true

   0.300000000000000

Parallel pool using the 'local' profile is shutting down.
>> ParforExample
Without A MatLabPool in operation.  No parfor loop
   0.008333333333333

fx >>
```

Fig. 5.1: Parfor-Loop Example Output

not maintained a different ultimate value for r would be obtained. Figure 5.1 demonstrates the operation of the parfor-loop compared to the for-loop equivalence in a sequential program.

```
t = [1,2,3,4,5];
r = 1;
matlabpool open 2
parfor i=1:5
    r = r / t(i);
end
disp(r)
matlabpool close
```

As the simplest form of parallel programming, some of the earliest work of this research utilised the 'parfor' command. For example the calculation of the coefficients used within the Diethelm-Chern algorithm (see Chapter 8). Later work required referrals to other iterations of any potential parfor-loop. In this case the parfor command would produce inaccurate results and therefore would not be appropriate. Consequently the parfor command did not feature in the final programs of this research.

5.3.2 Batch jobs

A batch job allows the user to send a program to another computing resource which then operates the program with the desired pool of workers. As mentioned in Section 5.2, the MatLab Distributed Computing Server should be installed if the batch operation is sent to a remote pool/cluster. By batching the program to a remote cluster, the user's computer or **'Client'** is able to continue operating without interference from the batched parallel program. As the Reader will see from Chapter 6 the parallel architecture used within this research operated a local parallel profile and therefore the batch command was not required.

5.3.3 Distributed Arrays and Single Program Multiple Data

Distributed and co-distributed arrays allow the manipulation/use of large data sets across a pool of workers. The 'distributed' command operates from the Client with segments of the array sent or created on the workers in the pool. The Client has no control over how the array is divided amongst the workers. By creating the distributed array directly on the worker pool the Client memory is not adversely affected so this form of parallelism is advantageous for large data sets [44]. An example program containing a distributed array is provided below:

```
w=distributed.eye(6,6);
disp(w)
spmd
    for i=1:6
        w(i,i)=w(i,i)*labindex;
    end
    getLocalPart(w)
end
```

The command 'distributed.eye' provides a distributed identity matrix. To view data on each of the LABs the command 'getLocalPart' is used. The user can access data from any element of the distributed array as though it was stored on their Client machine.

A co-distributed array is created on the workers, by the workers. Each worker can access any element of the array, however accessing an element which is not stored locally will increase communication/time to completion of program [44]. An example program containing a co-distributed array is provided below:

```
n=[1,2,3,4,5];
x=codistributed(n);
t=x*2;
getLocalPart(t)
```

The research in this thesis considered numerical methods with equispaced discretisation. However the use of a distributed array could allow more variation in the step sizes within the 'current' incarnation of a numerical method (see Section 3.2). These varied step sizes could be stored within a distributed array for division amongst workers in the pool.

Single Program Multiple Data (SPMD)

The distributed array example in the previous section utilised the SPMD command. The SPMD (Single Program Multiple Data) command is, as in the traditional sense (see Section 4.2.1), a single program which runs concurrently on multiple LABs using different data input. In the example the SPMD command allowed each LAB to operate upon its own portion of the distributed array using the same for-loop command. This example demonstrates how an SPMD can be structured to process large data sets, and to simultaneously operate time-expensive programs across a pool of LABs (see [44]).

In the case of the Runge-Kutta programs, and early work with ODE algorithms, the application of the SPMD command enabled each LAB to run the algorithm with a different step size. For example, LAB1 would have step size 0.1, LAB2 would use 0.05, LAB3 would use 0.025 and LAB4 would use 0.0125. The results of the separate algorithms were communicated back to a single LAB for application of an adapted Richardson Extrapolation (see Chapter 7).

Later work with SPMD took a 'block' approach (see Section 4.5). This mirrored earlier work of Diethelm [14] who undertook the same procedure on different platforms (FORTRAN and C, with MPI). Figure 5.2 illustrates the procedure for a 1000 step numerical method undertaken on a four-LAB pool. The 1000 steps are split into groups or 'blocks'. Each 'block' contains four steps, replicating the size of the MatLab pool. The program works through each block in sequence on the pool. The procedure continues until all blocks (and hence steps) have been completed. Please note: Although the initial problem eminates from a Client Computer (the Blue Computer in Figure 5.2) the Client could also be part of the four-worker pool. In this research

the Client is part of the MatLabPool, see Chapter 6.

Further information is available in Chapter 6 regarding the use of the SPMD command to restrict multithreading within MatLab programs.

5.4 Communication between LABs

In a standard parallel architecture it is customary to use some form of message passing interface (Section 4.5.1). In MPCT functions based upon the MPI-2 standard [51] have been built into the software. Sharma et al [58] indicate that generic tasks such as loading, starting, uploading and cleaning are all handled by the software. Functions are provided for point-to-point and broadcast operations [44], these are:

- labSend - Sends data to a specific LAB within the pool.
- labReceive - Receives data from a specific LAB within the pool. This command is required when the transmitting LAB uses labSend.
- labSendReceive - Sends data to another LAB in the pool and receives data from that same entity.
- labBroadcast - Sends data to all LABs in the pool.

In addition to the communication commands provided above, additional commands not utilised by this research are also provided [44], these are:

- The global communication functions used to concatenate (gcat), add (gplus) and operate (gop) across the pool.
- The labProbe command to ascertain readiness for comunication on a LAB.
- The labBarrier command to prevent further parallel work until all LABs reach the same point.

The use of the labSend/labReceive and the labBroadcast commands were investigated further to ascertain which was the quickest for the types of parallel progams operationalised in this research (see Chapter 6).

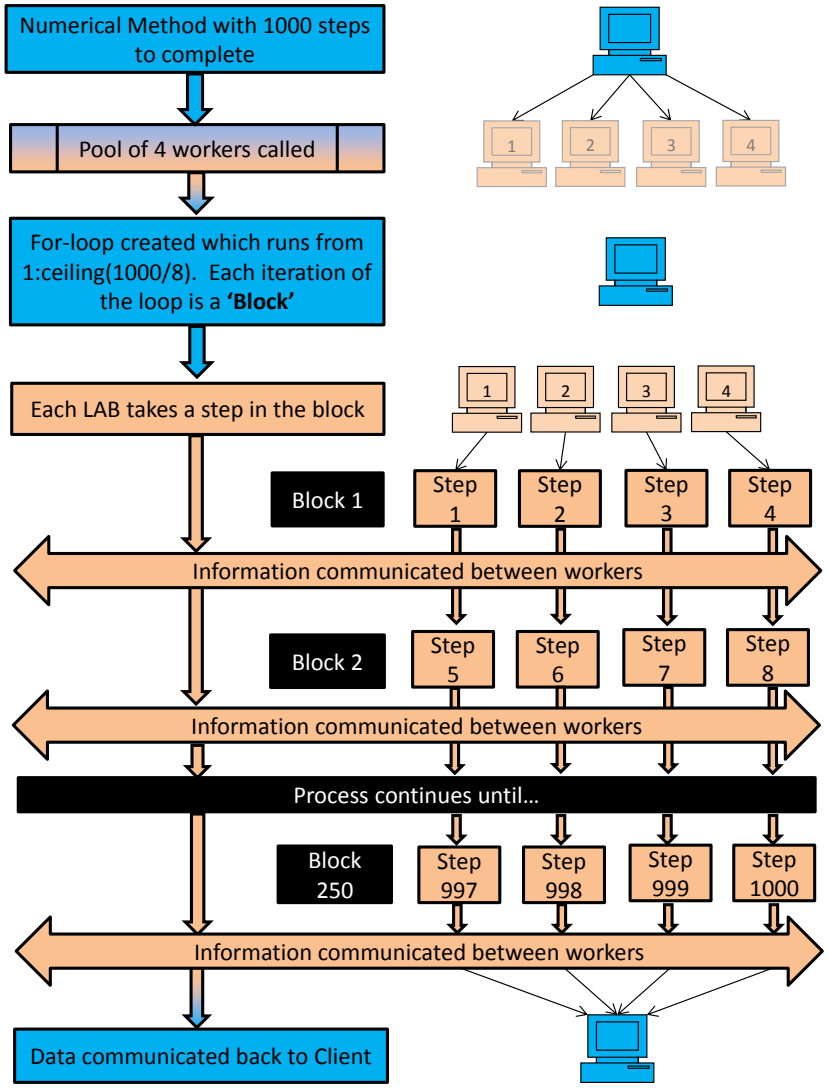

Fig. 5.2: Example of block programming

5.5 Tools to assess efficiency in MPCT

The traditional commands of tic/toc can be used to measure the time taken for a program to operate. The 'tic' command is placed at the beginning of the program with the 'toc' at the end. The call of the 'toc' command displays the total elapsed time in the 'Client' or controlling LAB. This form of time measurement was pursued in the course of this research.

In addition, traditional MatLab has a Profiler for assessing the areas of a program which take the most time. The Profiler can be executed via the 'Run and Time' option built into the MatLab Editor. Upon completion of the program the Profiler tool launches, providing a visualisation of the programs performance and highlighting where most time is spent in computation.

MPCT extends the Profiler to allow performance measurement of workers when operating on a communicating job. By 'communicating job' we are referring to the creation of "a single task that runs simultaneously on several workers, usually with different data" [45]. The Parallel Profiler does not lend itself to independent jobs i.e. where each worker performs different tasks. Additionally the Parallel Profiler will not operate if the prototype program uses high-level constructs e.g. parfor, SPMD.

5.6 Discussion Points

Following the research in this Chapter, a general understanding of MPCT has been established. It is evident that the valuable prototyping facilities of MatLab have been extended to MPCT providing easy access to parallel programming. As a result of this Chapter the following discussion points will be taken forward in this research:

- The parfor and SPMD constructs for parallel programming.
- The implicit multithreading built into MPCT and MatLab generally.
- The labSend/labReceive and labBroadcast commands for communicating between LABs.
- The tools of tic/toc and the Profiler which can be used to assess performance of parallel programs.

6. HARDWARE, SOFTWARE, AND DATA COLLECTION

6.1 Introduction and Objectives

Following the previous Chapters of this thesis, Chapter 6 will specify the methodolgy used during data collection. Implicit multithreading in MatLab Parallel Computing Toolbox (MPCT) is discussed for later experimentation in Chapters 7 - 10. Additionally this Chapter demonstrates attempts to reduce overheads through varying methods of communication between workers.

In summary, this Chapter aims to satisfy the following objectives:

- To inform the Reader of the equipment used in this research.
- To detail the methodology employed when collecting data.
- To report on implicit multithreading in MPCT and how multithreading can be controlled.
- To report on small scale experimentation with communication between workers in a parallel architecture.

6.2 Hardware and Software Specifications

6.2.1 Hardware

The Mathematics Department at the University of Chester has two multiprocessor computers referred to as 'Euler' and 'Abel'. Each of these computers has four-cores on each of the two sockets, thus providing an eight-worker environment used in this research. Euler/Abel does not utilise hyperthreading technology [37] as they were created prior to the reintroduction of hyperthreading into Intel processors (see Section 4.4.1).

Additionally a standard desktop computer (laptop) provides a parallel environment useful for developing prototype programs. Due to the hyperthreading technology within an Intel Processor (e.g. Intel Core i3) a standard

laptop will provide a four-worker parallel environment (see Section 4.4.1). However, as Chapter 7 demonstrates, the Euler/Abel architecture produced quicker results and consequently later work concentrated on Abel for all program execution.

6.2.2 Software

Following the discussion in Chapter 5 the software employed in this research was MatLab Parallel Computing Toolbox (MPCT). The MatLab Distributed Computing Server software was not utilised as the hardware used enabled discrete application without the need for batching to a remote cluster. Abel in the Mathematics Department was accessed remotely through PuTTY, a program which enables a SSH connection. SSH provides a secure connection between a client and server by encrypting the information passed between the computing resources over an insecure network [49]. This connection enabled the execution of MPCT directly on Abel within a terminal. Therefore the time taken for the constructed programs to run did not include any time-lags from buffering.

6.2.3 Benchmarking Data

Figure 6.1 provides benchmarking data for the standard Intel Core i3 laptop used in the research of Chapter 7. The Laptop has 4GB memory and a clock speed of 2.13 GHz. The Laptop operates on Windows 7 Home Premium (64-bit). The version of MPCT installed on the Laptop was R2014a. Figure 6.2 provides benchmarking data for Abel used in the course of this research. Abel has 7.8GB memory and a clock speed of 2.3GHz x 8. Abel operates on Ubuntu 12.04 (64 bit). The version of MPCT installed on Abel is R2012b.

6.3 Methodology for data collection

In Chapters 7 - 10 the same methodology for data collection was employed. Namely, the basic implementation involves the creation of four programs: a four-worker parallel program, an eight-worker parallel program, a sequential program with multithreading enabled, and a sequential program without multithreading for comparison purposes. Additional programs have been created which adapt this basic four-program structure. These extra programs allowed additional functionality e.g. the application of Richardson Extrapolation, the repetition of the program 13 times to assist in data collection

Machines used in benchmark (R2014b)

- Linux (64-bit) 3.47 GHz Intel Xeon = 6 threads, 24 GB memory
- Windows 7 Enterprise (64-bit) 3.47 GHz Intel Xeon = 6 threads, 24 GB memory
- Windows 7 Enterprise (64-bit) 2.7 GHz Intel Core i7 = 2 threads, 8GB memory
- Mac OS X Mountain Lion (64-bit) 2 GHz Intel Core i7 = 4 threads, 4GB memory
- Mac OS X Lion (64-bit) 2.66 GHz Intel Xeon = 12 threads, 16 GB memory
- Windows 7 Enterprise (64-bit) 2.66 GHz Intel Core 2 Quad = 4 threads, 4 GB memory
- Windows XP (32-bit) 2.4 GHz Intel Core 2 Quad = 4 threads, 3.48 GB memory

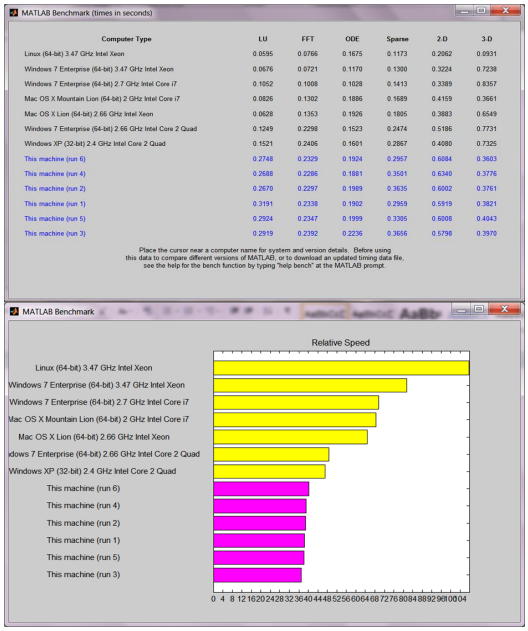

Fig. 6.1: Benchmark data from Matlab - Laptop

Machines used in Benchmark (R2012b)

- Linux (64-bit) 3.47 GHz Intel Xeon = 6 threads, 24 GB
- Windows7 Enterprise (64-bit) 3.47 GHz Intel Xeon = 6 threads, 24 GB
- Windows 7 Enterprise (64-bit) 2.7 GHz Intel Core i7 = 2 threads, 8GB
- Mac OS X Mountain Lion (64-bit) 2 GHz Intel Core i7 = 4 threads, 4 GB
- Mac OS X Lion (64-bit) 2.66 GHz Intel Xeon = 12 threads, 16 GB
- Windows 7 Enterprise (64-bit) 2.66 GHz Intel Core 2 Quad = 4 threads, 4 GB
- Windows XP (32-bit) 1.86 GHz Intel Core 2 = 2 threads, 2GB

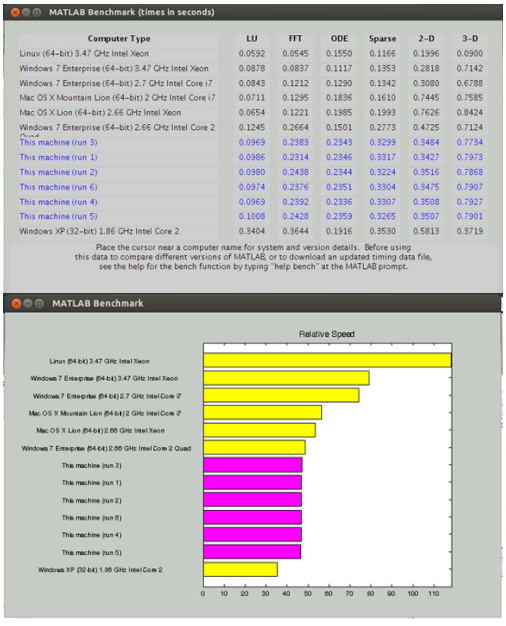

Fig. 6.2: Benchmark data from Matlab - Abel

(see later for explanation), or for experimentation with communication commands (see Chapter 9).

Each group of programs was executed thirteen times. The time for each program to execute was recorded using the tic/toc commands. The first result of the set of thirteen executions was discarded as this often performed much slower than the 'typical' execution time. This phenomenon is a result of the Just In Time (JIT) interpretation processes when MatLab is first opened; subsequent execution of the prototype programs are unaffected. This issue has also been experienced by other MatLab users (see reference [46]). The fastest and slowest results in the remaining twelve timings were discarded to remove outliers leaving ten results for averaging (mean). These average results were reported in subsequent chapters.

Within each of the numerical methods all variables, except step size and/or termination point, were kept constant e.g. value of fractional derivative, and other constants. However, in order to increase the size of the problem the step size h was decreased whilst maintaining the termination point for the numerical method i.e. the final point of the period of integration. This methodology was adopted in Chapters 9 - 10. Chapters 7 - 8 slightly altered this approach by maintaining the step size and by increasing the termination point 'Last'. The same effect was achieved in both approaches which was to increase the total number of steps the numerical method was evaluated over.

6.3.1 Efforts to obtain best performance

In order to obtain the best performance of the parallel and sequential programs all other computer operations were reduced to a minimum. In other words, no other software was in operation whilst the laptop was executing parallel programs. In the case of Abel other users were occasionally logged-into the server but not necessarily using software on the device. In order to reduce the chances of other users accessing Abel, the device was used mainly at weekends and evenings.

6.4 Opening and Closing the MatLab Pool

When creating a parallel program using MPCT an initial command to open and close a 'pool' of workers has to be called. The command 'matlabpool' or

No. of LABS	**Opening Pool** (*sec*)	**Closing Pool** (*sec*)
One LAB	5.591	3.285
Two LABs	5.877	3.281
Three LABs	6.053	3.278
Four LABs	6.265	3.295
Five LABs	6.615	3.310
Six LABs	6.964	3.341
Seven LABs	7.359	3.330
Eight LABs	8.025	3.347

Tab. 6.1: Opening and Closing the MatLab Pool

'parpool' [1] incurs communication overheads which are consistent depending upon the number of workers called into action. A program was created which allowed the user to open/close the MatLab Pool for a desired number of LABs. Once complete, the program would display a line of text indicating it had completed its task. The Table 6.1 provides the average execution times for the matlabpool/parpool commands on one to eight-worker pools (experiments conducted on Abel, see Section 6.2). The data indicates a small time increment as each additional LAB is added to the pool. In a small-scale problem the overheads incurred through opening the pool would be significant. Therefore the time taken for opening/closing the pool should be considered when developing prototype parallel programs.

6.5 Forms of Communication between Workers

Let us now consider communication between workers which has a bearing on future prototyping.

Two commands used for communicating between workers are labBroadcast and labSend/labReceive (see Section 5.4 in Chapter 5). The labBroadcast command sends a blanket message to all LABs with the data specified by the user. Targetted LAB-to-LAB communication is achieved using the labSend/labReceive commands.

Table 6.2 indicates the number of communication points for an example program where labBroadcast and/or labSend/labReceive are employed. By

[1] N.B. The 2014 version of MatLab Parallel Computing Toolbox indicates a change in command from the original 'matlabpool' to the new 'parpool' command. The data in Table 6.1 is for the original 'matlabpool' command.

'communication point' we are referring to: a labBroadcast which is sent to all workers in the pool (this would be seven communication points for an eight-worker pool), or for the labSend/labReceive commands which would be one communication point between the Transmitting LAB and the Receiving LAB. In Table 6.2 a problem with ten steps to complete is considered using three methods of communication. These three methods were:

- Program 'A' - uses a mixture of labBroadcast (when all LABs require the information) and labSend/labReceive (when only one/some of the LABs require the information). Data is sent to LABs irrespective of whether the data is needed i.e. if the LAB is active in the block or not.

- Program 'B' - uses the same mixture of labBroadcast and labSend / labReceive. Where labBroadcast is usually in use 'if' statements have been included to determine if the receiving LABs are active, if not a labSend command is used instead of labBroadcast. Additionally these 'if' statements allow for the acceptance of data on LAB4 which acts as the hub for reporting back to the User from the Pool.

- Program 'C' - uses labBroadcast command with all LABs irrespective of whether the data is needed by the receiving LAB or not. A series of 'if' statements are included to allow the acceptance of data on idle LABs.

No. of LABs	**Program 'A'**	**Program 'B'**	**Program 'C'**
4	25 (9)	21 (9)	30 (12)
5	28 (14)	24 (14)	40 (20)
6	36 (20)	30 (20)	50 (30)
7	42 (27)	33 (27)	60 (42)
8	50 (35)	38 (35)	70 (56)

Tab. 6.2: Number of communication points between LABs

The number in brackets refers to the number of communications points within one block of the ten step example (see Figure 6.3 to illustrate this data on a four-LAB pool).

From Figure 6.4 we can observe that Programs A and B perform marginally better over an increased number of steps than Program C where labBroadcast is solely used. Results are less pronounced on the four-LAB programs when compared to the eight-LAB equivalent. Considering the differences in execution time in conjunction with the number of communication points in

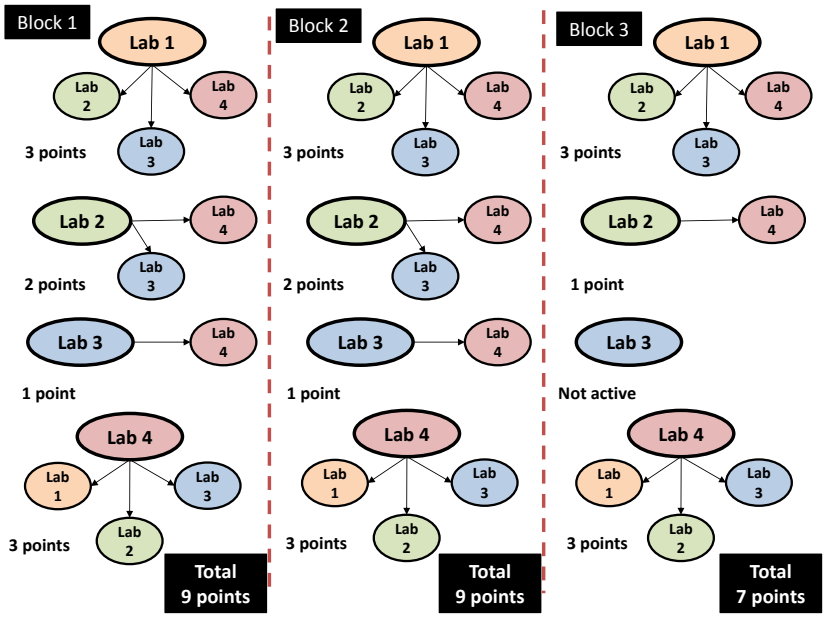

Fig. 6.3: Example illustrating the number of communication points in Program A

Table 6.2, the results are not surprising i.e. there are an increased number of communication points resulting in increased execution time. The slight difference between Program A and Program B could be accounted for when considering the increased lines of code required for 'smart' and targetted communication between the LABs. This data demonstrates only slight differences in time for the different communication mechanisms. The differences could become more pronounced over a larger MatLab Pool and therefore the careful use of labBroadcast should be a consideration for a user of MPCT.

6.6 Implicit Multithreading

Following discussion on multithreading in Chapter 4 it is now important to consider the implications of multithreading. MatLab implicitly utilises multithreading in matrix and element-wise operations within constructed programs [48]. Although implicit multithreading could provide advantages in sequential code it is, in fact, providing a type of parallel environment. When a multiworker pool is called in a multiprocessor architecture, each LAB has only one thread at its disposal. By comparing the multithreaded sequential programs to multi-worker parallel programs we are undertaking an unfair comparison. Although advantages should be gained in implicit multithreading, data in Chapter 9 demonstrates the negative impact of multithreading

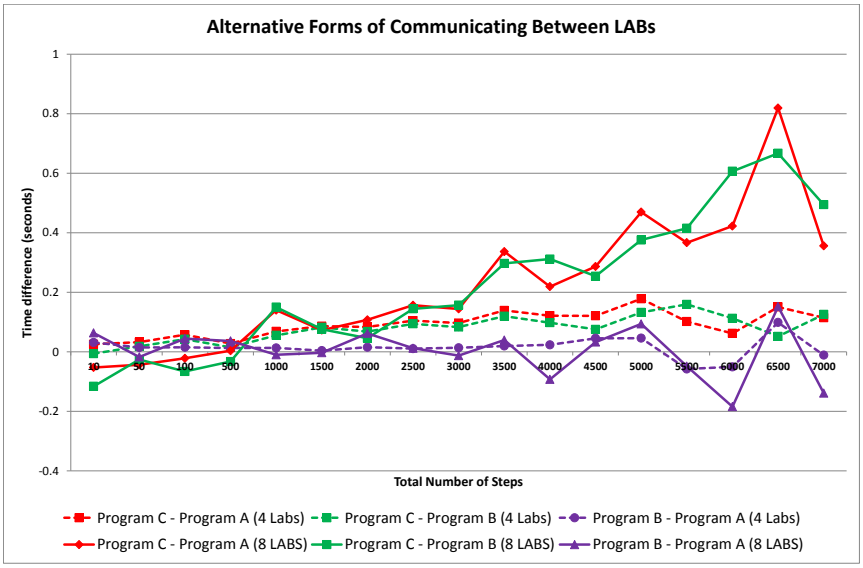

Fig. 6.4: Experiments with communication - differences between alternative communication commands

upon the execution time of the program. This result mirrors experience in the High Performance Computing sector as discussed in [39] where the Author turned off multithreading and discovered an improvement in performance.

To restrict multithreading in a sequential program, it is not sufficient to simply open a MatLab Pool with a single worker. Using the matlabpool/parpool command merely activates the required number of workers to action. However if a programmer calls a command such as SPMD the workers are activated and consequently the single LAB would then take control. The SPMD command was used within this research to restrict multithreading within a sequential program. As an alternative the command 'maxNumCompThreads' allows the programmer to control the threads directly. The 'maxNumCompThreads' command was additionally used with Lubich's Fractional Multistep Method to control threading before and after the SPMD (see Chapter 10).

6.7 Assessment of Performance

As discussed in Chapter 4 the assessment of performance can be undertaken in several ways. The method chosen in this research was the calculation of the theoretical and practical benefit of implementing a numerical method in parallel. The 'practical benefit calculations' involved the average time taken for the programs to run at a defined number of steps. A percentage increase/decrease was calculated by comparing the parallel program performance against the sequential program (without multithreading). For the 'theoretical benefit calculations', Excel spread sheets were devised for every program under analysis. Each line of code was broken down to calculate the number of actions undertaken to complete the program (see Section 4.6.2 for information). The spread sheets allowed the Researcher to input a different step size and termination point, providing an estimated total number of actions for the program. For the examples used in Chapters 7 and 10 an exact solution was known. Consequently a program was created which compared the exact and calculated solution using the numerical method. Once a given error tolerance was achieved the step size was reported and used in the spread sheet. The same step size was used in the program to calculate the practical benefit.

For the examples in Chapters 8 and 9 the solutions involved the Mittag-Leffler function. As the Mittag-Leffler function is not an exact solution a step size was chosen which provided a large problem for the parallel programs to

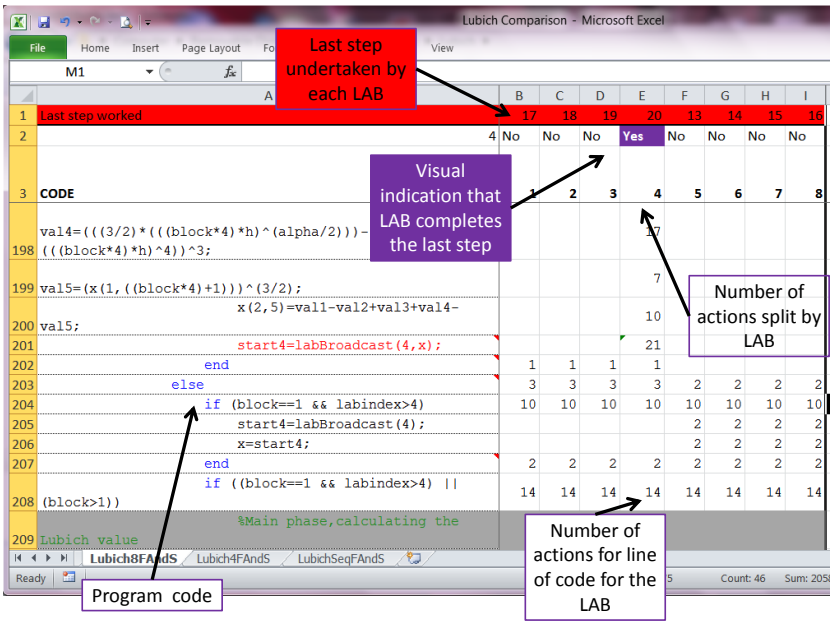

Fig. 6.5: Extract from a theoretical calculation spread sheet

execute. Again this step size was used in the spread sheet for theoretical benefit calculations and an average execution time was calculated for the practical benefit calculations.

An extract from an example Excel spread sheet is provided in Figures 6.5 and 6.6.

In Section 5.5 the MatLab Profiler was discussed as a method for analysing prototype programs. Unfortunately the Profiler does not analyse certain aspects of code, in particular where there are for-loops or 'if' statements. The programs created in this research substancially used for-loops and 'if' statements. Consequentally the Profiler was unable to assist in assessing the efficiency of prototyped parallel programs.

6.8 Discussion Points

In this Chapter the hardware and software used within this research were detailed. Information was also provided regarding the data collection methodology and analysis technique. As a consequence of this Chapter the following items will be carried forward to subsequent chapters:

- The hardware and software specification used within the research.

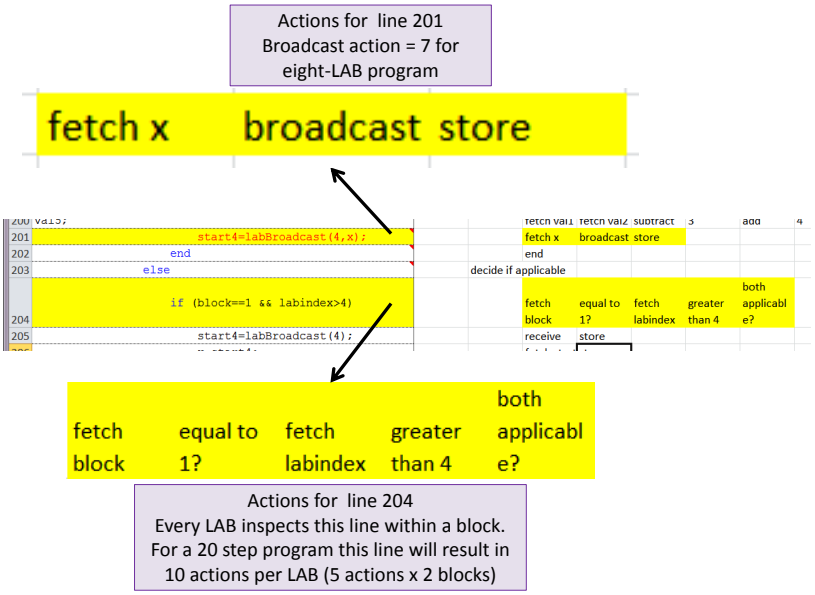

Fig. 6.6: Example line of code and the number of actions

- The implications of implicit multithreading within sequential programs as parallel program comparitors.
- The impact upon execution time of opening and closing the MatLab Pool.
- How the number of actions can be calculated using the Excel spread sheet.

7. ORDINARY DIFFERENTIAL EQUATIONS: RUNGE-KUTTA METHOD

7.1 Introduction and Objectives

Following the work in Chapter 5, this Chapter will demonstrate initial attempts at prototyping parallel programs in MatLab Parallel Computing Toolbox (MPCT). To begin with consideration is given to numerical methods for Ordinary Differential Equations (ODEs). After experimentation with six numerical methods given in Lambert [40] a Runge-Kutta Method for ODEs was implemented in parallel and compared to a sequential equivalent. The application of an adapted Richardson Extrapolation (RE) Scheme, utilising four approximations to the solution, has been provided. To summarise, this Chapter aims to satisfy the following objectives:

- Demonstrate how the Runge-Kutta Method can be implemented in MPCT.
- Adapt the Richardson Extrapolation (RE) Scheme to enable the calculation of more accurate results using four approximate solutions.
- Implement the Adapted RE Scheme in MPCT and compare results to sequential equivalent.
- Provide analysis of the performance for the resultant parallel and sequential programs on different architectures.
- Demonstration of the impact of multithreading on sequential program performance.

7.2 Runge-Kutta Method for Ordinary Differential Equations

As mentioned in the Introduction to this Chapter, initial experimentation with MPCT began with considering six numerical methods provided in the book by Lambert [40]. These six methods were:

- Linear Multistep Method x3 (Section 3.3.1)
- Predictor-Corrector (Section 3.3.3)
- Runge-Kutta algorithm x2 (Section 3.3.4)

Lambert [40] demonstrates that the last of his Runge-Kutta Methods was convergent for all step sizes (h). As a consequence the remainder of the MPCT prototyping concentrated upon this Runge-Kutta Method (see Section 7.2.1).

7.2.1 Runge-Kutta Method: The Algorithm

To assist the Reader a brief reminder of the Runge Kutta Method is provided. Equation 7.1 provides the general formula for the Runge-Kutta Method,

$$y_{j+1} = y_j + h \sum_{i=1}^{s} b_i K_i, \tag{7.1}$$

where $i = 1, 2, ...s$ and:

$$K_i = f\left(t_j + hc_i, y_j + h \sum_{n=1}^{s} a_{i,n} K_n\right). \tag{7.2}$$

Unlike the Linear Multistep Methods (Section 3.3.1) the Runge-Kutta Methods are multistage methods i.e. use 'off-step' points to approximate the solution rather than approximations obtained at previous steps. K_i are the 'off-step' approximations to the solution at $t_j + hc_1, \ldots, t_j + hc_s$ where s is the order of convergence.

The particular example in Lambert [40] of the Runge-Kutta Method is provided in Equations 7.3, 7.4, and 7.5:

$$y_{n+1} - y_n = \frac{h}{2}(K_1 + K_2) \tag{7.3}$$

where

$$K_1 = f(t_n, y_n) \tag{7.4}$$

$$K_2 = f\left(t_n + h, y_n + \frac{1}{2}hK_1 + \frac{1}{2}hK_2\right) \tag{7.5}$$

7.2.2 Adapted Richardson Extrapolation

Following on from Section 3.5.1 a variation upon the original Richardson Extrapolation Scheme (Equation 3.115) was achieved to accommodate four approximations to the solution $y(t_n)$. These four approximations were at time steps h, $\frac{h}{2}$, $\frac{h}{4}$ and $\frac{h}{8}$ and results in the following equations:

$$\begin{aligned} y(t_n) &= z_n + h^p Q + O\left(h^{p+1}\right) && (7.6)\\ y(t_n) &= w_n + (0.5h)^p Q + O\left(h^{p+1}\right) && (7.7)\\ y(t_n) &= s_n + (0.25h)^p Q + O\left(h^{p+1}\right) && (7.8)\\ y(t_n) &= g_n + (0.125h)^p Q + O\left(h^{p+1}\right) && (7.9) \end{aligned}$$

Ignoring the $O(h^{p+1})$ terms in Equations 7.6 - 7.9 above and mulitplying 7.7 by 2^p gives:

$$2^p y(t_n) = 2^p w_n + h^p Q \tag{7.10}$$

Multiplying 7.8 by 4^p and rearranging:

$$4^p y(t_n) = 4^p s_n + h^p Q \tag{7.11}$$

Multiplying 7.9 by 8^p and rearranging:

$$8^p y(t_n) = 8^p g_n + h^p Q \tag{7.12}$$

Calculating 7.12 - 7.11 - 7.10 + 7.6:

$$\begin{aligned} 8^p y(t_n) - 4^p y(t_n) - 2^p y(t_n) + y(t_n) &= (8^p g_n + h^p Q) - (4^p s_n + h^p Q) - (2^p w_n + h^p Q) + (z_n + h^p Q)\\ 8^p y(t_n) - 4^p y(t_n) - 2^p y(t_n) + y(t_n) &= 8^p g_n + h^p Q - 4^p s_n - h^p Q - 2^p w_n - h^p Q + z_n + h^p Q\\ y(t_n)\left(8^p - 4^p - 2^p + 1\right) &= 8^p g_n - 4^p s_n - 2^p w_n + z_n \end{aligned}$$

This results in the final Adapted Richardson Extrapolation equation:

$$y(t_n) = \frac{8^p g_n - 4^p s_n - 2^p w_n + z_n}{8^p - 4^p - 2^p + 1} \tag{7.13}$$

7.3 Implementation in MatLab

The local parallel architecture (standard laptop, see Section 6.2) provided a maximum of four workers. To allow comparisons across architectures, efforts concentrated upon a four-worker pool which could be executed locally (standard Intel Core i3 laptop) and on Euler/Abel housed in the Mathematics Department (see Section 6.2).

The Runge-Kutta Method was implemented in MPCT using the SPMD command (see Section 5.3.3). The SPMD command allowed separate processors to implement the algorithm using different step sizes. In order to later implement the Adapted Richardson Extrapolation Scheme (Equation 7.13) four step sizes were chosen which were multiples of each other. The step size h was fixed for each worker: LAB1 used 0.1, LAB2 used 0.05, LAB3 used 0.025, and LAB4 used0.0125. To increase the problem size the termination variable was increased (this is called 'Last' in the programs).

The exact solution to the example initial value problem was known (see Section 7.4). It was therefore possible to compare exact to approximate results. Where Richardson Extrapolation was applied the extrapolated results were compared. However for programs where no extrapolations was applied each set of results were compared to the exact solution for accuracy. This form of implementation therefore resulted in increased number of actions for the programs without Richardson Extrapolation.

The following programs were created and used in analysing the performance of prototype Runge-Kutta programs in MPCT:

- **RevisedSeqLambertEx6VariableLastv2** - Opens the MatLab pool for one worker however multithreading is in operation. Undertakes the Runge-Kutta Method four times with the step sizes h, $\frac{h}{2}$, $\frac{h}{4}$, and $\frac{h}{8}$.

- **RevisedSeqLambertEx6VariableLastRE** - Same application of the MatLab pool and multithreading as RevisedSeqLambertLambertEx6VariableLastv2. The Runge-Kutta Method is operated four times with the step sizes h, $\frac{h}{2}$, $\frac{h}{4}$, and $\frac{h}{8}$. Additionally the Adapted Richardson Extrapolation is applied using the results from the four step sizes.

- **NEWRevisedSeqLamberEx6VariableLastv2** - Opens the MatLab pool for one worker. Undertakes the Runge-Kutta Method four times with the step sizes h, $\frac{h}{2}$, $\frac{h}{4}$, and $\frac{h}{8}$. Also uses the SPMD command to

ensure only a single worker is in operation at an equivalent point to parallel implementation in ParLambertEx6VariableLastONLY.

- **NEWRevisedSeqLamberEx6VariableLastRE** - Same application of the MatLab Pool and SPMD command as NEWRevisedSeqLamberEx6VariableLastv2. Again the Runge-Kutta Method is operated four times with the step sizes h, $\frac{h}{2}$, $\frac{h}{4}$, and $\frac{h}{8}$. Additionally the Adapted Richardson Extrapolation Scheme is applied using the results from the four step sizes.

- **ParLambertEx6VariableLastONLY** - Uses the SPMD command to run four different step sizes on four workers. For example: LAB1 uses step size 0.1, LAB2 uses step size 0.05, LAB3 uses step size 0.025, and LAB4 uses step size 0.0125.

- **ParLambertEx6REVariableLastv2** - Same parallel implementation as ParLambertEx6VariableLastONLY. All workers send results to LAB1 using the labSend/Receive command. LAB1 applies the Adapted Richardson Extrapolation Scheme.

Additional programs were created for a two-worker and eight-worker pool. These programs were not used in the rest of this Chapter but are available on the CD-ROM for the Reader. In particular the eight-worker parallel program with Richardson Extrapolation provides an additional adaptation to the orignal scheme. To summarise, the additional programs are:

- ParLambertExample62Labs - Runge-Kutta Method where only two workers are in operation within the pool. The SPMD command is used so each worker can take a different step size i.e. LAB1 uses h step size defined by the User, LAB2 therefore uses step size $\frac{h}{2}$.

- ParLambertExample6_8Labs - Runge-Kutta Method implementation utilising a pool of 8 workers. The SPMD command is used so that each worker can take a different step size i.e. LAB1 uses h step size defined by the User, subsequent LABs will use the step size $\frac{h}{2^{(\text{Labindex}-1)}}$ where Labindex is the number of the worker in the pool.

- ParLambertExample6RE2Labs - Same as ParLambertExample62Labs but with the additional application of the standard Richardson Extrapolation Scheme (see Section 3.5.1).

- ParLambertExample6RE8Labs - Same as ParLambertExample6_8Labs but with the additional application of an alternative Adapted Richardson Extrapolation Scheme. The alternative Scheme uses eight approximations to the solution.

An annotated version of the ParLambertEx6REVariableLastv2 program is provided in Appendix A.

7.4 Runge-Kutta Method: Results from MatLab

To proceed, the example inital value problem to be solved was provided by Lambert [40] and is,

$$y' = f(t, y), \tag{7.14}$$

where $y = [u, v]^T$, and

$$f(t, y) = \left[v, \frac{v(v-1)}{u}\right]^T \tag{7.15}$$

with initial conditions $y(0) = \left[\frac{1}{2}, -3\right]^T$.

As a slight variation on the intended methodology descibed in Section 6.3, each Runge-Kutta program was executed twelve times. No serious anomilies in execution times were experienced in the reported results in this chapter e.g. the opening of the MatLab Pool which occurs on first use of the MatLab software. The top and bottom execution time results were also omited to remove outlying data. The remaining ten execution times were averaged (mean) and this data was used in subsequent tables and figures in this section. Figure 7.1 and Figure 7.2 provide the execution times of the parallel and sequential programs created in MPCT for the Runge-Kutta Method. 'Local' refers to the standard laptop available to the Researcher, and Abel is the multiprocessor server in the Mathematics Department (see Section 6.2.1). Figure 7.1 provides the execution times for the sequential (no multithreading) and parallel programs where no Richardson Extrapolation is applied. Figure 7.2 provides the execution times for the sequential (no multithreading) and parallel programs where the Adapted Richardson Extrapolation is applied.

From Figure 7.1 it can be observed that the Abel implementation over four-workers performed the quickest. The sequential program executed on Abel performed poorly when the value of 'Last' was increased. By increasing the value of 'Last' the period of integration would be extended and hence the problem size would be increased for the numerical method. Concentrating on the data in Figure 7.1, the sequential programs produced slower results

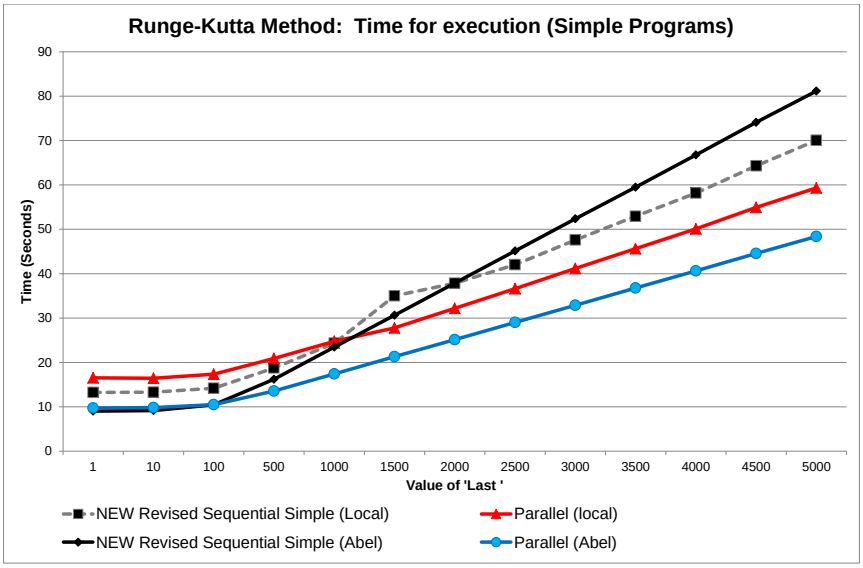

Fig. 7.1: Runge-Kutta Method - Execution Times (no Richardson Extrapolation)

as the problem size increased. An increased discrepency between the sequential programs and the parallel programs is observed as the value of 'Last' increased. For 5000 steps it is observed the difference between the sequential and parallel programs executed on Abel was approximately 33 secs.

In Figure 7.2 the Abel implementation over four-workers performs the quickest yet again. In general the results in Figure 7.2 demonstrated a better performance than programs where no Richardson Extrapolation (RE) had been implemented. This result was not surprising given the single point of comparison between the exact and extrapolated results (the programs without RE had to conduct this comparison an additional three times!) With the Local implementation a 'Tipping Point' was achieved for the parallel program around a 'Last' value of 4000. Again the increased execution time of the sequential programs is demonstrated as the problem size increases.

Using the formula Speedup $(n) = \frac{T(1)}{T(n)}$ (Speed Up equation - Equation 4.1, see Section 4.6.1) a ratio of sequential to parallel execution time can be obtained. Figure 7.3 provides the ratio results.

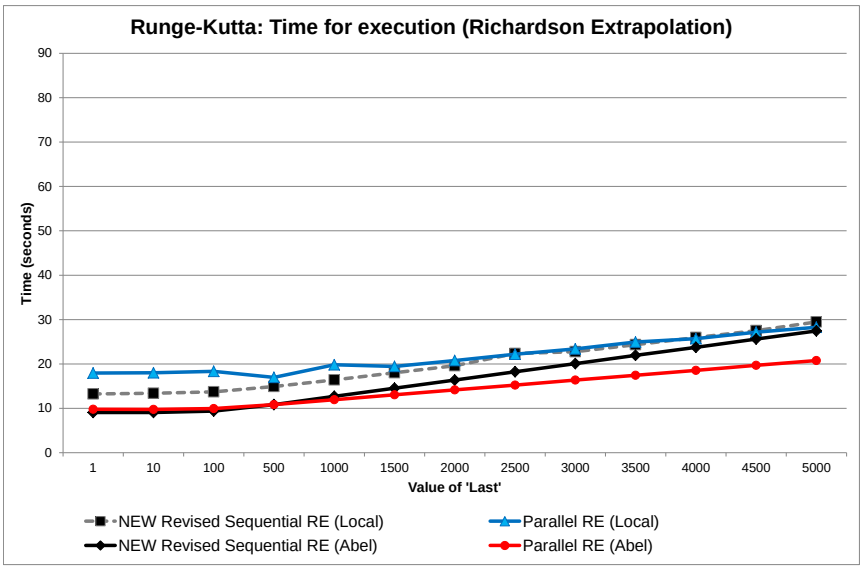

Fig. 7.2: Runge-Kutta Method - Execution Times (with Richardson Extrapolation)

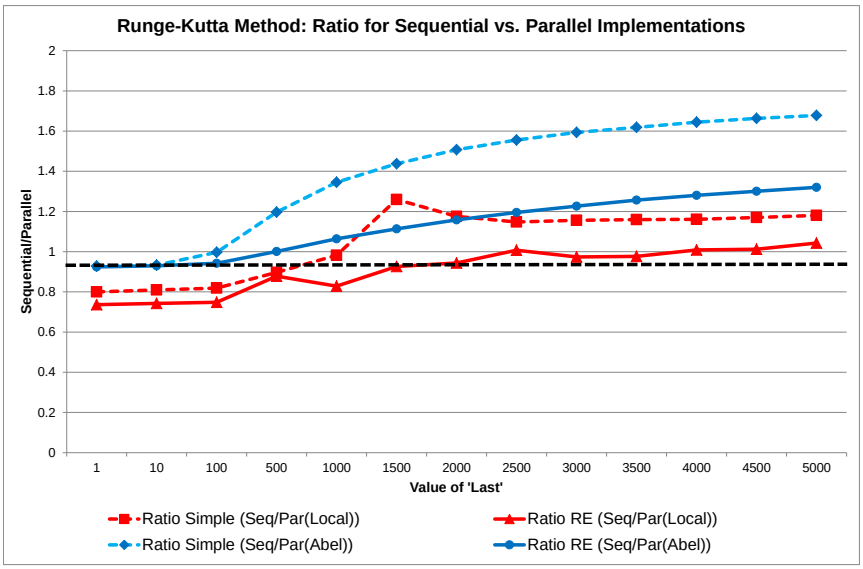

Fig. 7.3: Runge-Kutta Method - Ratio of Execution Times (Sequential vs. Parallel Implementation)

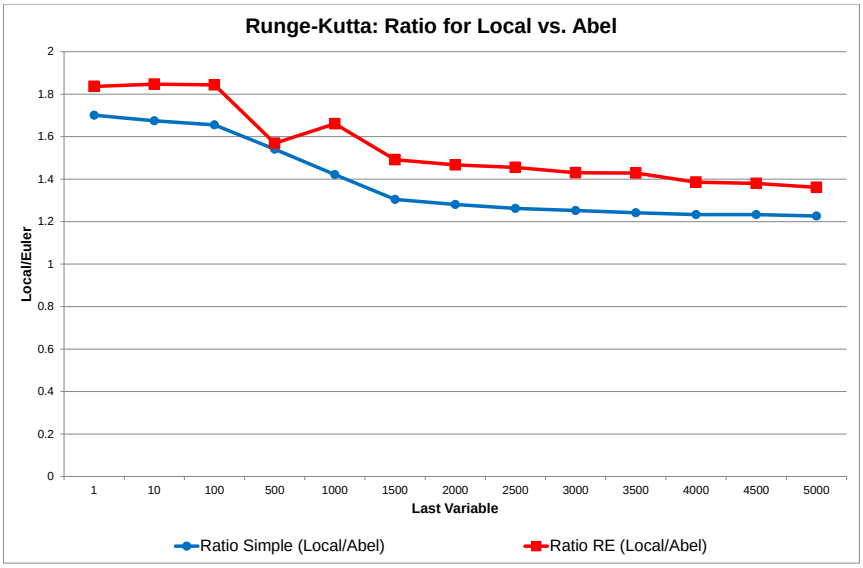

Fig. 7.4: Runge-Kutta Method - Ratio of Execution Times (Local vs. Abel Implementation)

It is observed in Figure 7.3 that the programs without Richardson Extrapolation almost achieve a speed up ratio of 1.8. In other words the sequential program took nearly twice as long to execute as the parallel program over four workers. Although the speed up ratios were positive results the scalability was poor. Additionally data in Figure 7.4 demonstrated that the Abel architecture does not sustain performance over increased number of steps. In fact neither Local nor Abel achitectures sustain parallel programming performance.

Sequential Programs - Impact of Multithreading

Figure 7.5 provides the execution times for the sequential programs with and without multithreading on the Local architecture. The time for programs to execute was effected by restricting multithreading. It can be observed from Figure 7.5 that the sequential program without RE and multithreading was over 30 seconds slower than it multithreaded counterpart when the value of 'Last' was 5000. Figure 7.6 provides the ratio for the sequential programs execution time. As the value of 'Last' increased the ratio for the sequential programs decreased, yet again highlighting the increasing execution time of

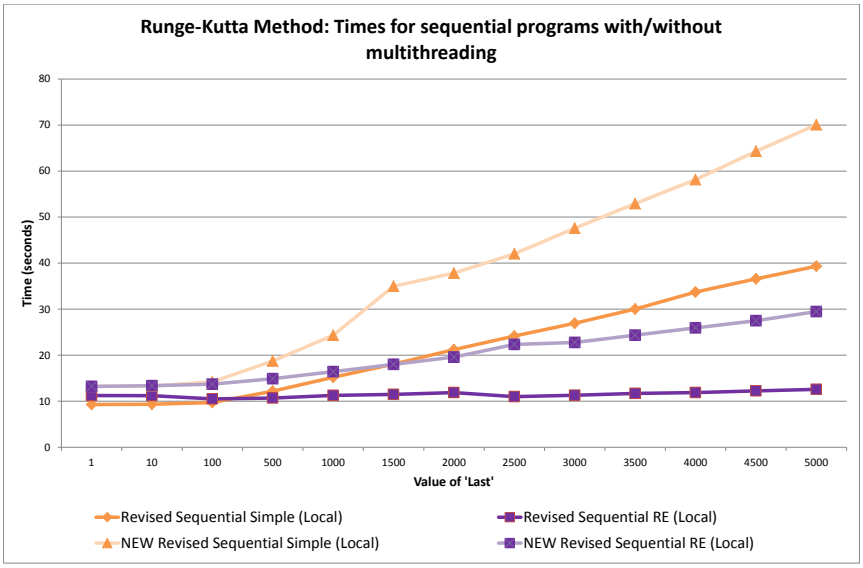

Fig. 7.5: Runge-Kutta Method - Time of Execution (Sequential Program with and without Multithreading)

the non-multithreading programs.

7.4.1 Runge-Kutta Method: Data Tables

For completeness Table 7.1 and Table 7.2 provide the (mean) average execution times for the sequential (no multithreading) and parallel programs.

7.5 Theoretical vs Practical Benefit Calculations

Table 7.3 summarises the results of the theoretical versus practical benefits of parallel implementation for the Runge-Kutta Method. One of the sequential programs was adapted to ascertain the step size which would achieve an error tolerance of 10^{-6}. The total number of steps ('No. of steps') for this step size is displayed in Table 7.3 for programs with and without the Adapted Richardson Extrapolation. The mean average execution time ('Time') is also displayed in the Table for the number of steps which will achieve the desired level of accuracy. Following Section 4.6.2 the 'No. of actions' were calculated

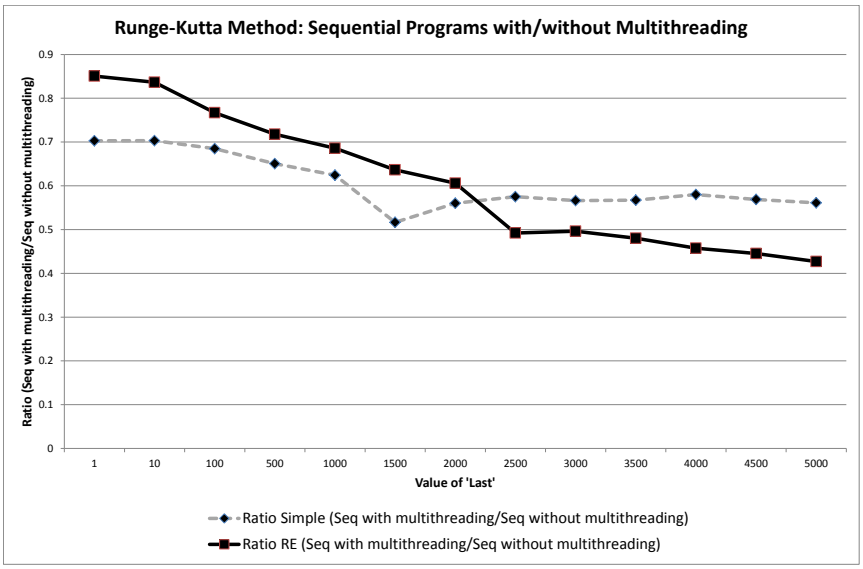

Fig. 7.6: Runge-Kutta Method - Ratio of Execution Times (Sequential Program with and without Multithreading)

'Last' Value	Sequential (Local)	Sequential (Abel)	Parallel (Local)	Parallel (Abel)
1	13.216	9.011	16.518	9.710
10	13.294	9.158	16.425	9.809
100	14.185	10.433	17.329	10.468
500	18.732	16.213	20.877	13.548
1000	24.322	23.441	24.762	17.422
1500	35.000	30.624	27.786	21.301
2000	37.840	37.891	32.185	25.133
2500	42.029	45.122	36.618	29.006
3000	47.584	52.367	41.170	32.869
3500	52.930	59.500	45.640	36.759
4000	58.155	66.778	50.100	40.620
4500	64.292	74.110	54.944	44.562
5000	70.071	81.183	59.341	48.385

Tab. 7.1: Runge-Kutta Method - Average Execution Time (no Adapted Richardson Extrapolation)

'Last' Value	Sequential (Local)	Sequential (Abel)	Parallel (Local)	Parallel (Abel)
1	13.221	9.039	17.956	9.778
10	13.376	9.069	18.019	9.755
100	13.703	9.366	18.321	9.938
500	14.900	10.826	16.967	10.817
1000	16.406	12.685	19.809	11.927
1500	18.012	14.521	19.447	13.039
2000	19.580	16.373	20.746	14.139
2500	22.336	18.218	22.175	15.239
3000	22.768	20.051	23.385	16.348
3500	24.337	21.922	24.924	17.441
4000	25.939	23.750	25.708	18.550
4500	27.488	25.607	27.158	19.685
5000	29.473	27.425	28.269	20.769

Tab. 7.2: Runge-Kutta Method - Average Execution Time (with Adapted Richardson Extrapolation)

for each of the MPCT programs and provided in Table 7.3. (* This result is for the highest number of actions which is always LAB4. The data in brackets refers to LAB1 which has the least number of actions.)

Program	No. of steps	No. of actions	Time (Local)	Time (Abel)
Sequential no RE	2500	3,592,783	16.676	13.350
Sequential RE	30	38,111	13.510	9.047
Parallel no RE	2500 (LAB1)	1,805,057* (230,066)	24.751	12.098
Parallel RE	30 (LAB1)	16,725* (4,674)	20.850	9.827

Tab. 7.3: Results to obtain error tolerance

From Table 7.3 the theoretical benefit of implementing the program in parallel would result in a 49.8% decrease in actions for the programs without the Adapted Richardson Extrapolation. However the practical benefit of parallel implementation is not favourable and results in a 48.4% increase in execution time on the local architecture. On Abel the same programs result in a 9.4% decrease in execution time.

Considering the Adapted Richardson Extrapolation Scheme implementation in the sequential and parallel programs, the data is also unfavourable.

The theoretical benefit of implementation of the parallel programs results is a 56.1% decrease in actions. Again, practically the local architecture results in a 54.3% increase in execution time. With the Abel architecture the practical benefit is a 8.6% increase in execution time.

Both of the theoretcial vs. practical benefit calculations above highlight the load balance problem with the constructed Runge-Kutta parallel programs. For example: if LAB1 is operating over 100 steps, LAB2 is operating over 200 steps, LAB3 is operating over 400 steps, and LAB4 is operating over 800 steps. The operated number of steps of LABs 1 - 3 combined are still less than LAB4's effort. Figure 7.7 illustrates the impact of the chosen step sizes for a four-worker pool. In Table 7.3 the quoted 'number of actions' refer to the worker with the most number of steps and therefore the most work. For example, the number of actions for the parallel program with no Adapted Richardson Extrapolation (labelled 'Parallel no RE') is 1,805,057. This refers to the work of LAB4. In comparison LAB1 performs 230,066. A percentage decrease in actions of 87.3% from LAB4.

7.5.1 FLOP Calculation

Following the information presented in Section 4.6.3 a calculation of the floating point operations has been undertaken for the sequential and parallel prototype programs. In both the sequential and parallel case, the FLOP calculation result were the same - the order of arithmatic was $O(N)$ where N was the total number of steps in the numerical method.

7.6 Discussion Points

This Chapter has provided initial attempts at constructing parallel programs in MPCT. Theroetically the implementation of the Runge-Kutta Method in parallel would be beneficial, for an accuracy of 10^{-6} a minimum gain of 49.1% decrease in actions is obtained. However practical results have highlighted the poor load balancing which occurs through the use of step sizes h, $\frac{h}{2}$, $\frac{h}{4}$, and $\frac{h}{8}$. Predictably the poor load balance has affected the execution time of the parallel programs leading to a 54.3% increase in execution time on the local architecture.

The research in this Chapter has provided an Adpated Richardson Extrapolation Scheme for use with four approximations to the solution. The

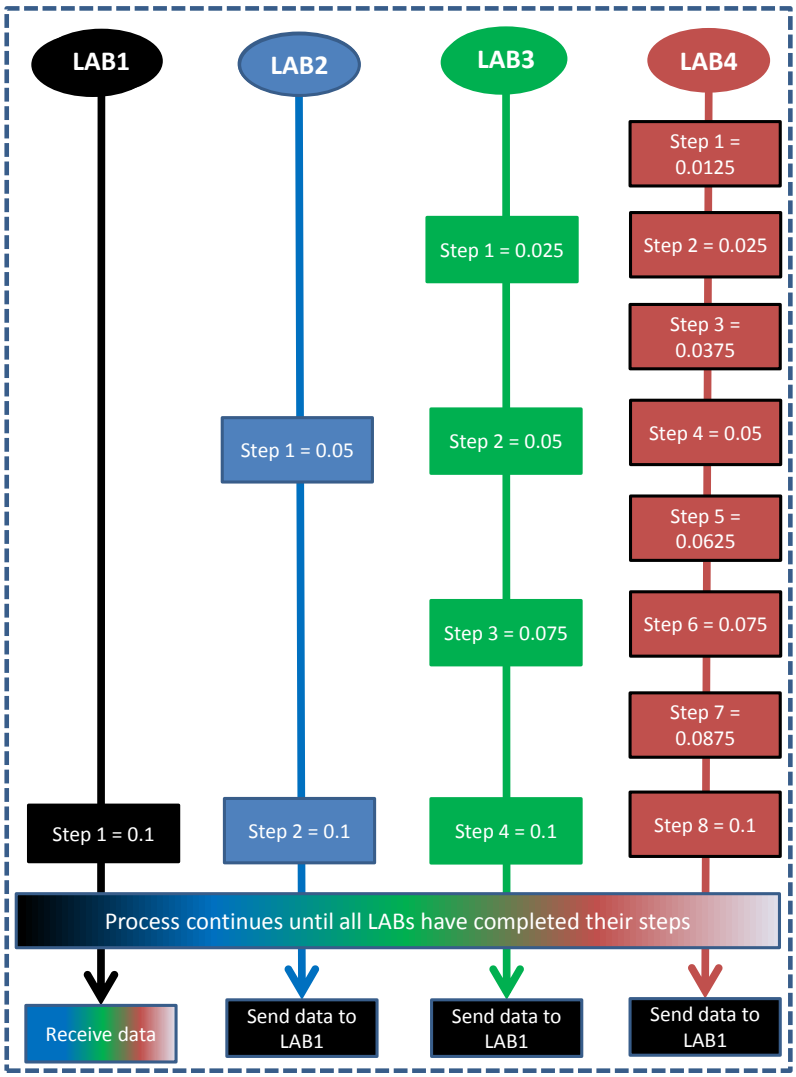

Fig. 7.7: Runge-Kutta Method - Example of poor load balance

Adapted Richardson Extrapolation Scheme enables the solution of the initial value problem to a higher degree of accuracy using a larger step size than ordinary results would require.

Following the research in this Chapter, the following points will be taken forward in this thesis:

- The use of the SPMD command to allow each worker to operate difference step sizes.
- The Adapted Richardson Extrapolation Scheme for four approximations to a solution.
- The use of Abel Architecture to obtain the quickest execution of the MPCT programs. This result is unsurprising given the increase in memory provided in this architecture's specification.

8. DIETHELM-CHERN ALGORITHM

8.1 Objectives

The Diethelm-Chern Algorithm was discussed in Chapter 3. This Chapter will illustrate how the Diethelm-Chern Algorithm can be implemented in MatLab Parallel Computing Toolbox (MPCT) in sequential and parallel form. As an expansion on Chapter 7 the Diethelm-Chern Algorithm will also be presented in this Chapter as an eight-worker parallel program. An analysis of the time taken to run the MPCT programs will be presented to ascertain when parallel programs perform better than the sequential counter-part. Additionally theoretical and practical calculations will demonstrate potential gains in using parallel programs using MPCT for the Diethelm-Chern Algorithm.

To summarise this Chapter aims to satisfy the following objectives:

- To provide sequential and parallel programs for the Diethelm-Chern Algorithm in MPCT.
- To expand the parallel program constructs to an eight-worker pool using MPCT.
- To analyse performance of the MPCT programs for the Diethelm-Chern Algorithm.
- To present sequential programs with and without multithreading.

8.2 Diethelm-Chern: The Algorithm

From Diethelm [13], and discussed in Section 3.4.3 (equations 3.110 and 3.107) the Diethelm-Chern Algorithm can be expressed as:

$$y_j = \frac{1}{\theta_{0j} - \left(\frac{j}{n}\right)^{\alpha} \Gamma(-\alpha)\,\lambda} \left(\frac{j}{n}^{\alpha} \Gamma(-\alpha)\, f_j - \sum_{k=1}^{j} \theta_{kj} y_{j-k} - \frac{1}{\alpha} y_0 \right)$$

Where the weights θ_{kj} are defined by:

$$\alpha\left(1-\alpha\right)j^{-\alpha}\theta_{kj}=\begin{cases}-1 & \text{when k = 0}\\ 2k^{1-\alpha}-\left(k-1\right)^{1-\alpha}-\left(k+1\right)^{1-\alpha} & \text{when k=1,2,3,...,j-1}\\ \left(\alpha-1\right)k^{-\alpha}-\left(k-1\right)^{1-\alpha}+k^{1-\alpha} & \text{when k=j}\end{cases}$$

8.3 Implementation in MatLab

Initial attempts at creating parallel programs to implement the Diethelm-Chern Algorithm concentrated upon the parfor-loop and its application to the weight equations θ_{kj}. These initial attempts were thought unsuccessful however the impact of implicit multithreading within the sequential program was yet to be fully understood. Consequently the parfor-loop could be explored further for the Diethelm-Chern weights.

After initial work with the parfor-loop the SPMD command was explored. Following the work of Chapter 7 it was necessary to adapt the parallel programs to provide a better load balance when using the SPMD command. To this end a block approach was utilised. By 'block' approach we are referring to the division of the total number of steps into smaller groups of equal size to the number of available workers (see Section 4.5). Each worker would therefore be undertaking the same number of steps. Within a block each worker takes one consecutive step depending upon the identified LAB Number. Although the first LAB (LAB1) can complete its work, subsequent LABs require information from LAB1 and any preceding LABs in the block. For example: LAB4 will require the results from LABs1-3 in order to complete its algorithm, LAB5 will require data from LABs 1-4 etc. In the Diethelm-Chern Algorithm each LAB has to use the LabBroadcast command to provide its data to others in the pool. Figure 8.1 provides a diagramatical representation of how the block structure operates. The 'block' approach was utilised by Diethelm for the Fractional Adams Method [14] and will be discussed further in Chapter 9.

To perform a relatively comparable performance the sequential program included the opening of the MatLab Pool with one worker and the SPMD command to restrict multithreading.

The following programs were created in MPCT and are included on the CD attached to this thesis:

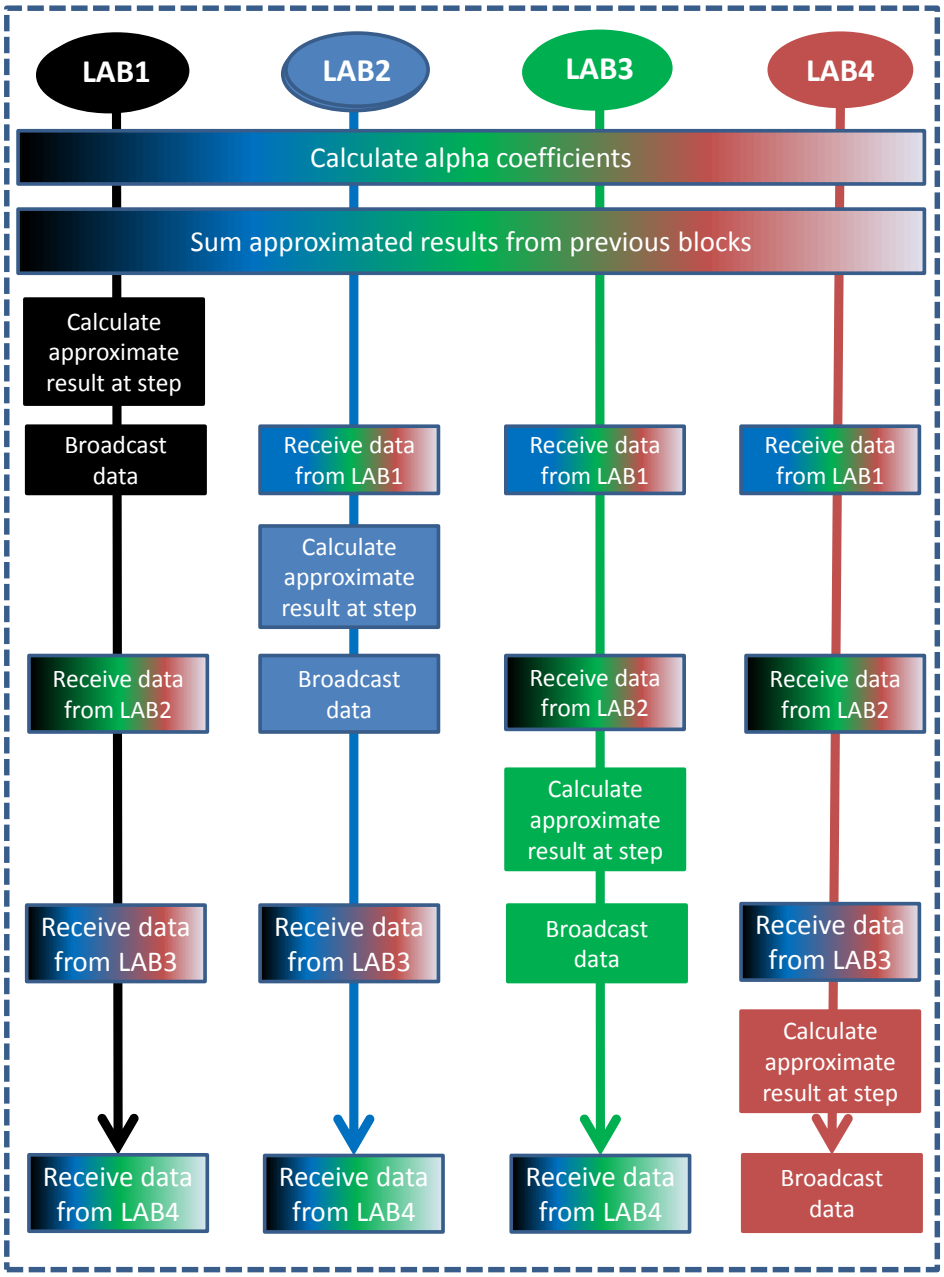

Fig. 8.1: Diethelm-Chern Algorithm: Block Method

- **FractionalProgSequentialv2INLOOP** - Sequential program for the Diethelm-Chern Algorithm. The program opens a MatLab Pool with only one worker but allows multithreading to operate. The program executes thirteen times for averaging purposes.

- **NEWFractionalProgSequentialv2THREADSINLOOP** - Sequential program for the Diethelm-Chern Algorithm. The program includes an open of the MatLab Pool and SPMD command to include the same elements as the parallel program and restrict multithreading. The program operates thirteen times for averaging purposes.

- **RevisedFractionalProgParallelv2INLOOP** - Four-worker parallel program for the Diethelm-Chern Algorithm. The program operates a 'block' method utilising the SPMD command so workers can take a different step in the block. The program operates in a loop to rerun data thirteen times, providing an execution time at the end of each loop for averaging purposes.

- **RevisedFractionalProgParallelv2_8LabsINLOOP** - Eight-worker parallel program for the Diethelm-Chern Algorithm. The program operates a 'block' method ultilising the SPMD command so workers can take a different step in the block. The program operates in a loop to rerun data thirteen times, providing execution time at the end of each loop for averaging purposes.

The program 'RevisedFractionalProgParallelv2_8LabsINLOOP' has been annotated and included in Appendix B.

In addition programs were created to experiment with methods of communication between workers in the Pool. The details of the experiments were included in Section 6.5 in Chapter 6. These additional programs are also included on the CD with this thesis. The programs from Chapter 6 are:

- EXPERIMENT2_6RevisedFractionalProgParallelv2INLOOP - This program is referred to as 'Program A' in Section6.5. This program uses labSend/labReceive or LabBroadcast whether the worker is active or idle. The program runs thirteen times for averaging purposes.

- EXPERIMENT2_7RevisedFractionalProgParallelv2INLOOP - This program is referred to as 'Program B' in Section 6.5. This program uses labSend/labReceive or labBroadcast however a series of 'if' statements are included to use appropriate communication with the pool

i.e. if all LABs are active then labBroadcast is used otherwise labSend/labReceive.

- EXPERIMENT3_5RevisedFractionalProgParallelv2INLOOP - This program is referred to as 'Program C' in Section 6.5. This program uses labBroadcast in all communication and includes a series of 'if' statements to accept data when LAB is idle.

8.4 Results in MatLab

An example equation was chosen for use in the prototype programs, this is provided in Equation 8.1.

$$D^{0.5}y(t) = -0.5y(t) \tag{8.1}$$

where $y(0) = 1$ and $0 \leq t \leq$ Last and 'Last' is defined by the User of the program. In the experiments below the variable 'Last' has been increased to provide a greater period of integration. The total number of steps in a unit is given by N. Therefore the step size is given as $h = \frac{1}{N}$. Throughout the research in this Chapter the step size h remained constant at 0.1.

Figure 8.2 provides the time taken for each program to complete for total number of steps 1000 to 7000. The graph illustrates the increasing execution time of the sequential program (without multithreading) as the total number of steps increases. Noticeably, the four and eight-worker programs mirror each others performance.

Figure 8.3 provides the ratio of sequential (without multithreading) to parallel program performance. The ratio is the equation from Section 4.6.1 where Speedup $(n) = \frac{T(1)}{T(n)}$. Looking at Figure 8.3 in conjunction with Table 8.1 the 'Tipping Point' at which the parallel program perform quicker than the sequential counter-part is achieved around 500 steps for the four-worker program and 1000 steps for the eight worker program. Both the four and eight-worker programs achieve a speed up of approximately 2 by 7000 steps.

Sequential programs with/without multithreading

As with Section 7.4 the sequential program data is provided to illustrate the impact of implicit multithreading. Figure 8.4 provides the execution time for the sequential programs with and without multithreading. The Figure clearly demonstrates the growth in the sequential program where multithreading

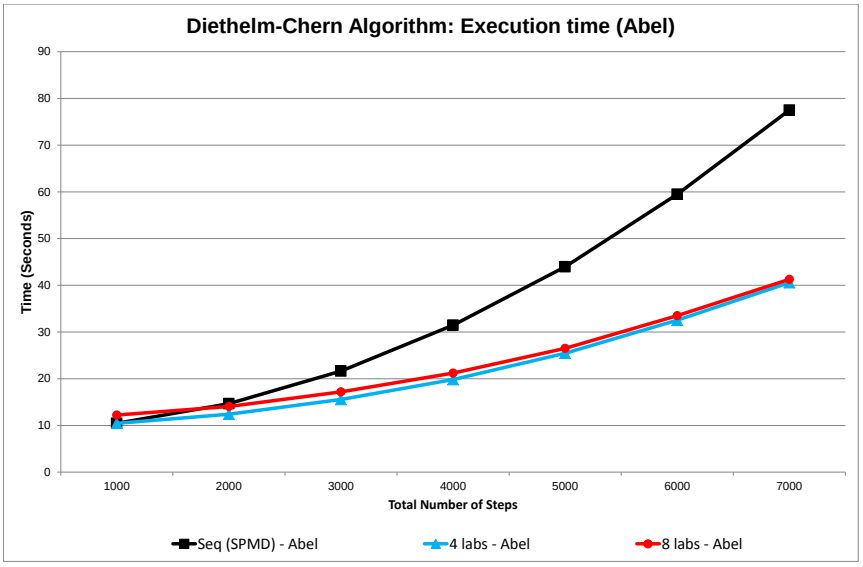

Fig. 8.2: Diethelm-Chern Algorithm: Execution Time (seconds)

was not included is far greater than its multithreaded comparator. The discrepency between the programs is also demonstrated in Figure 8.5 where the ratio has achieved below 0.3 for a total number of steps equaling 7000.

8.4.1 Diethelm-Chern Algorithm: Data table

Table 8.1 provides the mean average execution times at various step sizes including those provided in the Figures 8.2 and 8.3. All programs were run on Abel the eight-worker computer in the Mathematics Department (see Section 6.2). The program referred to as 'Sequential' does not include multithreading.

8.5 Theoretical/Practical Benefits Calculations

Table 8.2 provides the Theoretical and Practical Benefit calculations for step size 0.0004.

From Table 8.2 the theoretical benefit of parallel implementation results in a 22.5% decrease in actions for both the four-worker and eight-worker programs. The practical benefit calculations demonstrate a 21.9% and 12.7%

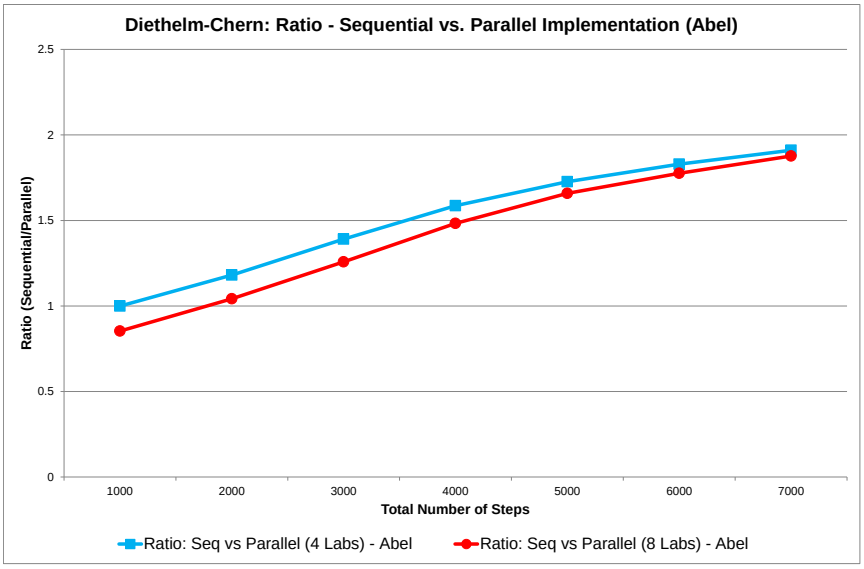

Fig. 8.3: Diethelm-Chern Algorithm: Ratio of sequential vs parallel program execution time

No. Steps	Sequential	Parallel (4 LABs)	Parallel (8 LABs)
50	9.046	9.747	11.492
100	9.049	9.742	11.470
150	9.076	9.731	11.590
200	9.106	9.769	11.528
250	9.124	9.773	11.581
300	9.177	9.808	11.551
500	9.395	9.920	11.693
750	9.842	10.161	11.961
1000	10.439	10.447	12.228
2000	14.653	12.411	14.066
3000	21.628	15.546	17.192
4000	31.456	19.836	21.216
5000	43.949	25.453	26.497
6000	59.489	32.525	33.506
7000	77.490	40.569	41.290

Tab. 8.1: Diethelm-Chern Algorithm: Execution Times (seconds)

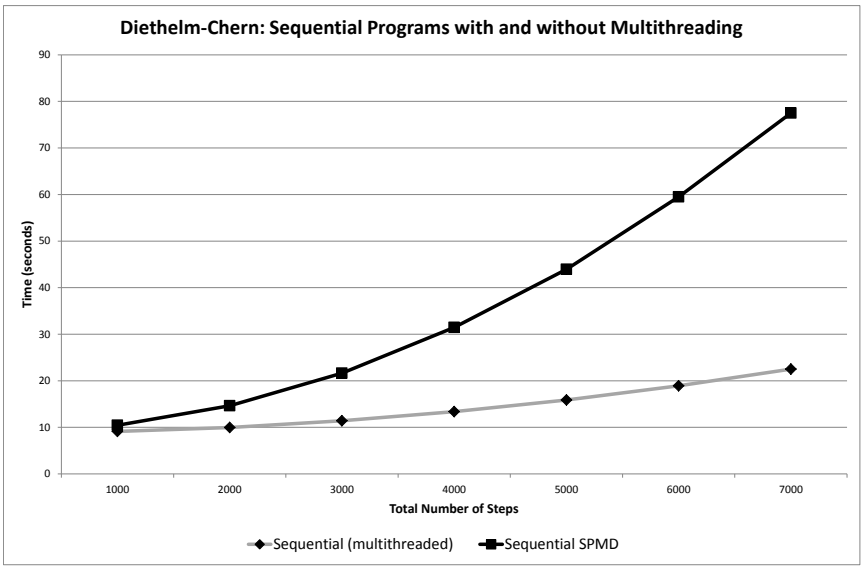

Fig. 8.4: Diethelm-Chern Algorithm: Execution time of sequential programs (seconds)

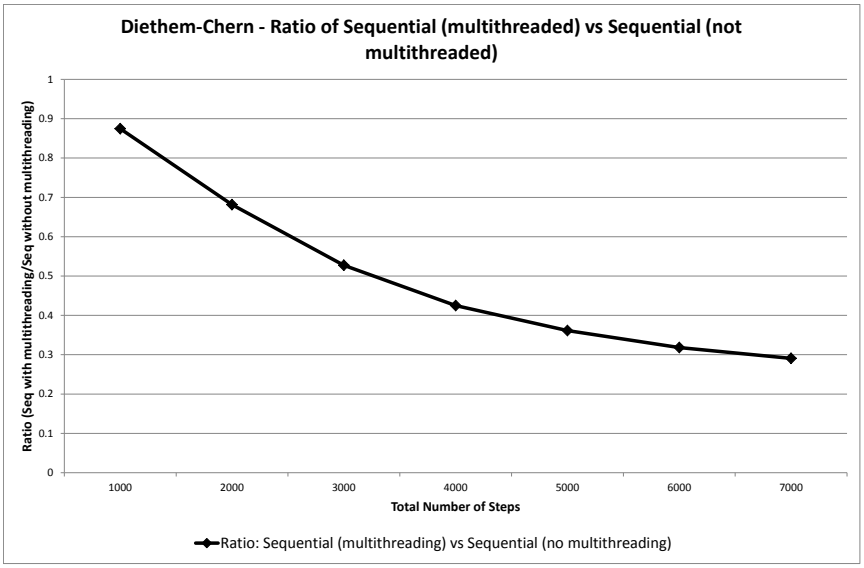

Fig. 8.5: Diethelm-Chern Algorithm: Ratio of sequential multithreaded and sequential non-multithreaded programs

Program	No. of steps	No. of actions	Time (sec)
Sequential	2500	194,015,043	17.857
Parallel (4 LABs)	2500	150,287,606 (150,261,949)	13.942
Parallel (8 LABs)	2500	150,271,463 (150,238,649)	15.594

Tab. 8.2: Diethelm-Chern Algorithm: Theoretical vs. Practial Benefit Calculations

decrease in execution time for the four-worker and eight-worker programs respectively. Table 8.2 also helps to explain the similar execution times experienced in Figure 8.2 as the number of actions required to complete the algorithm are remarkably similar. Unlike the Runge-Kutta Method Programs (see Chapter 7) the load balance between the workers is more favourable. The figure in brackets represent the worker with the lowest number of actions within the Pool.

8.6 FLOPs

An assessment of workload in the prototype programs can be assessed through the number of floating point operations. For the Diethelm-Chern programs (sequential and parallel) the order of arithmetic was $O(N^2)$ where N was the total number of steps in the method.

8.7 Discussion Points

From the research in this Chapter it is observed that the eight-worker parallel programs can be beneficial but as yet are slightly slower than the four-worker equivalent for the Diethelm-Chern program. The load balance problem experienced with the Runge-Kutta Method programs has now been addressed through the division of labour into 'blocks'. The labBroadcast command has been investigated further in Section 6.5 and demonstrated marginal increases in execution time when compared to the labSend/labReceive command.

Following the research in this Chapter the following points will be taken forwards:

- The use of the SPMD command for 'block' programming.
- The creation of eight-worker programs in MPCT.

9. FRACTIONAL ADAMS METHOD

9.1 Introduction and Objectives

In Section 3.4.2 the Fractional Adams Method (FAM) was provided and an example given. To continue the parallel programming work, the Fractional Adams Method was implemented in MatLab Parallel Computing Toolbox (MPCT) mirroring the method of 'parallelisation' given in Diethelm's paper [14]. In Paper [14] a 'block' approach was utilised (Section 4.5) i.e. the total number of steps (N) required to complete the method was divided into groups equal to the size of the MatLab Pool or as close as possible ($\lceil \frac{N}{\text{No. of workers}} \rceil$).

To summarise, this Chapter will aim to satisy the following objectives:

- To provide a set of programs in MPCT which utilise the block method of parallelisation, mirroring the work of Diethelm [14].
- To demonstrate the effect of multithreading upon the sequential program performance.
- To assess performance of the parallel programs resulting from the MPCT implementation.
- To compare performance of the MPCT parallel programs with the programs created by Diethelm which utilise C++ and MPI [14].

9.2 Fractional Adams Method: The Algorithm

To begin we recap the algorithm for the Fractional Adams Method as described in Chapter 3 (Section 3.4.2 and Baleanu et al [4]) and repeated here for the Reader. The Fractional Adams Method (FAM) comprises of two components: a 'Predictor' and a 'Corrector'.

Predictor

The first component of the FAM predicts the value of y at step k,

$$y_k^P = \sum_{j=0}^{\lceil\alpha\rceil-1} \frac{t_k^j}{j!} y_0^{(j)} + h^\alpha \sum_{j=0}^{k-1} b_{j,k}^* f\left(t_j, y_j\right), \tag{9.1}$$

where all previous values $y_1, \ldots, y_j$ have already been calculated and the initial condition y_0 has been provided.

The weights $b_{j,k}^*$ are defined as:

$$b_{j,k}^* = \frac{(k-j)^\alpha - (k-1-j)^\alpha}{\Gamma\left(\alpha+1\right)} \tag{9.2}$$

Corrector

The second component to the FAM uses the *'Predictor'* solution y_k^P within the *'Corrected'* value y_k:

$$y_k = \sum_{j=0}^{\lceil\alpha\rceil-1} \frac{t_k^j}{j!} y_0^{(j)} + h^\alpha \left(\sum_{j=0}^{k-1} b_{j,k} f\left(t_j, y_j\right) + b_{k,k} f\left(t_k, y_k^P\right) \right) \tag{9.3}$$

The weights $b_{j,k}$ are defined as:

$$b_{j,k} = \frac{1}{\Gamma\left(2+\alpha\right)} \begin{cases} \left((k-1)^{1+\alpha} - (k-\alpha-1)\, k^\alpha \right) & \text{when j=0} \\ \left((k-j+1)^{1+\alpha} + (k-j-1)^{1+\alpha} - 2\,(k-j)^{1+\alpha} \right. & \text{when } 1 \le j \le k-1 \\ 1 & \text{when j=k} \end{cases} \tag{9.4}$$

Please note: The application of the 'Corrector' element of the FAM Algorithm can be applied multiple times. However, the subsequent programming and application only consider the case where the corrector is only applied once. This mirrors the paper by Diethelm [14] to enable comparison of the computer program performance.

9.3 Fractional Adams Method: Implementation in MatLab

As indicated in the Introduction to this Chapter, the FAM implementation in MPCT mirrors that of Diethelm in [14]. In this paper, Diethelm uses a 'block' approach whereby the total number of steps are divided into smaller groups of equal size to the pool of workers. To implement the scheme the calculation of the predictor and corrector formulae are split into two phases. Phase one calculates the parts of the formula which do not rely on values which are currently being calculated by other LABs in the current block. After a series of communications between workers (the labBroadcast command is utilised in the MPCT programs) Phase two concludes the calculation of the predictor and corrector formulae using the newly received data.

Where the block calculation $\frac{\text{total number of steps}}{\text{Number of workers}}$ results in a non-integer, the total number of blocks is rounded up. For example: if the total number of steps is 750 and eight workers are in the MatLab Pool, 94 blocks would be required to complete the calculations. The last block would only operate on workers 1-6. The remaining two workers 7 and 8 would be idle. In such situations idle workers will not accept labBroadcast communications from active workers resulting in a message indicating the data was disregarded. To ascertain the impact of labBroadcast where there are idle workers in the last block, a set of programs was created which contained additional 'if statements'. These programs allowed idle workers to accept labBroadcast from active workers.

In the implementation, the calculation of the weights $b_{j,k}$ and $b^*_{j,k}$ was not undertaken within the parallel aspects of the program. Diethelm [14] indicates that the time taken for the weights to be calculated is "less than half a second" when the number of steps is 10^6.

The following list provides a brief description of the MPCT programs for the Fractional Adams Method (FAM). Programs **in bold** are used for further analysis in the remainder of the Chapter. The differences between the parallel programs listed below are summarised in Figure 9.1.

- **ABMegDSeqv3** - Sequential version of the FAM which opens a MatLab Pool for one worker. Multithreading is utilised in the program. The program operates in a loop which recalculates the results thirteen times, displaying the time taken for the program to operate at the end of each iteration.

- **ABMegDSeqTHREADSv2** - Sequential version of the FAM which

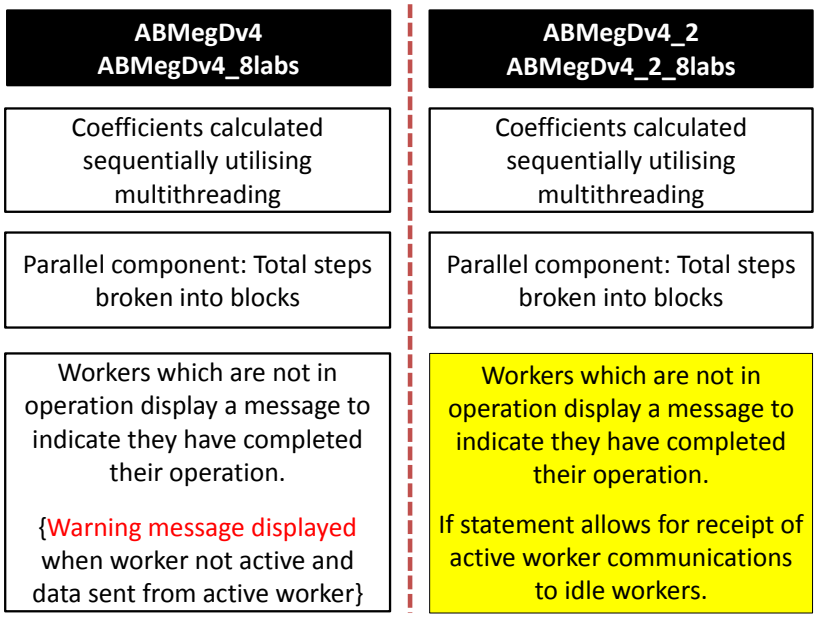

Fig. 9.1: Fractional Adams Method - Structural differences

incorporates the SPMD command. The SPMD command is needed to ensure only one LAB is in operation at a point equivalent to the parallel program comparators (Section 6.6 in Chapter 6). The program operates in a loop which recalculates the results thirteen times, displaying the time taken for the program to operate at the end of each iteration.

- **ABMegDv4** - Parallel version of the FAM utilising four workers. The program operates in a loop which recalculates the results thirteen times, displaying the time taken for the program to operate at the end of each operation.

- **ABMegDv4_8labs** - Parallel version of the FAM utilising eight workers. The program operates in a loop which recalculates the results thirteen times, displaying the time taken for the program to operate at the end of each iteration.

- **ABMegDv4_2** - Parallel version of the FAM utlising four workers. A series of 'if statements' allow for the acceptance of data from 'active' workers when a worker is idle. The program operates in a loop which recalculates the results thirteen times, displaying the time taken for the program to operate at the end of each iteration.

- **ABMegDv4_2_8labs** - Parallel version of the FAM utilising eight

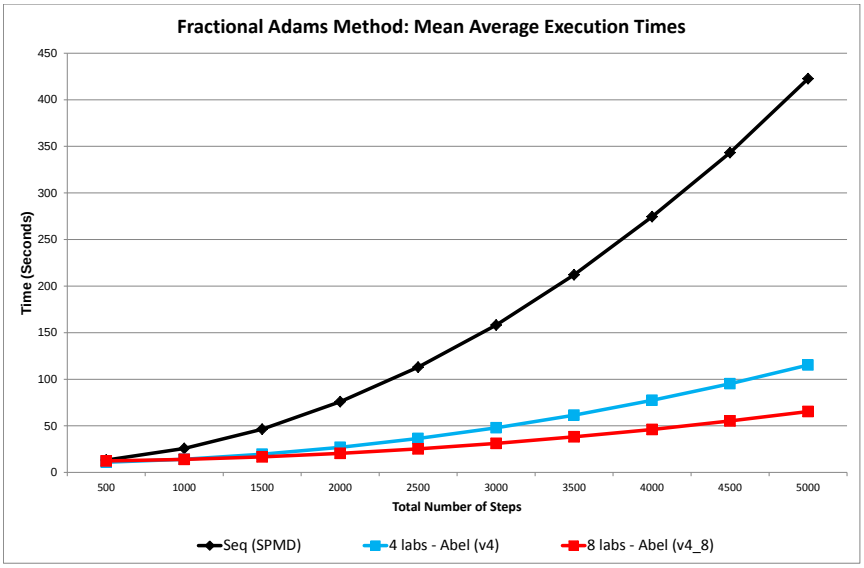

Fig. 9.2: Fractional Adams Method - (Mean) Average Execution Times

workers. A series of 'if statements' allow for the acceptance of data from 'active' workers when a worker is idle. The program operates in a loop which recalculates the results thirteen times, displaying the time taken for the program to operate at the end of each iteration.

An annotated version of the 'ABMegDv4' program is provided in Appendix C.

9.4 Results from MatLab

Figure 9.2 illustrates the execution times of:

- ABMegDSeqTHREADSv2
- ABMegDv4 - Executed on four workers.
- ABMegDv4_8 - Executed on eight workers.

Let us now compare the performance of the parallel programs against the sequential equivalent. Figure 9.3 provides the ratio of sequential program

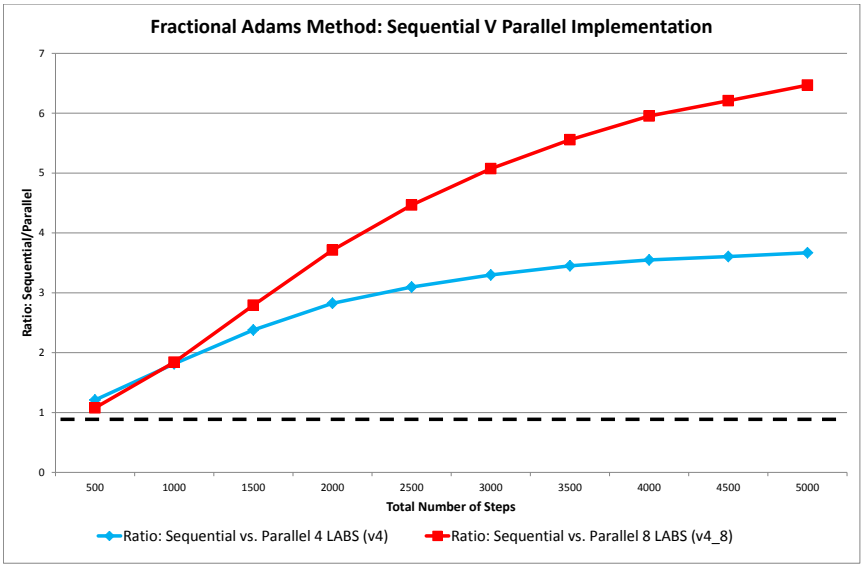

Fig. 9.3: Fractional Adams Method - Ratio of (Mean) Average Execution Times

against the parallel program using the Speed Up Equation 4.1 (see Section 4.6.1) i.e. Speedup $(n) = \frac{T(1)}{T(n)}$. In Figure 9.2 it can be seen that the 'Tipping Point' (see Section 4.6.1) where the parallel programs perform faster than the sequential equivalent is before 500 steps. The eight-worker program performs better than the four worker program around 1000 steps. No benefit is obtained when using the eight worker program over a small number of steps. This observation can be attributed to the increased communication overheads with the larger pool. Once the problem size becomes large the effect of the pool overheads are mitigated by increased efficiencies from using the eight worker pool. From Figure 9.3 the ratio of sequential to eight worker is around 6.5 for 5,000 steps.

9.4.1 Sequential Programs and Multithreading

As discussed in Section 6.6 multithreading is built into the MatLab language. Controls have to be placed in the sequential program to ensure only one thread (and therefore one worker) is in operation before using the program as a comparator to parallel prototypes. Following on from previous chapters

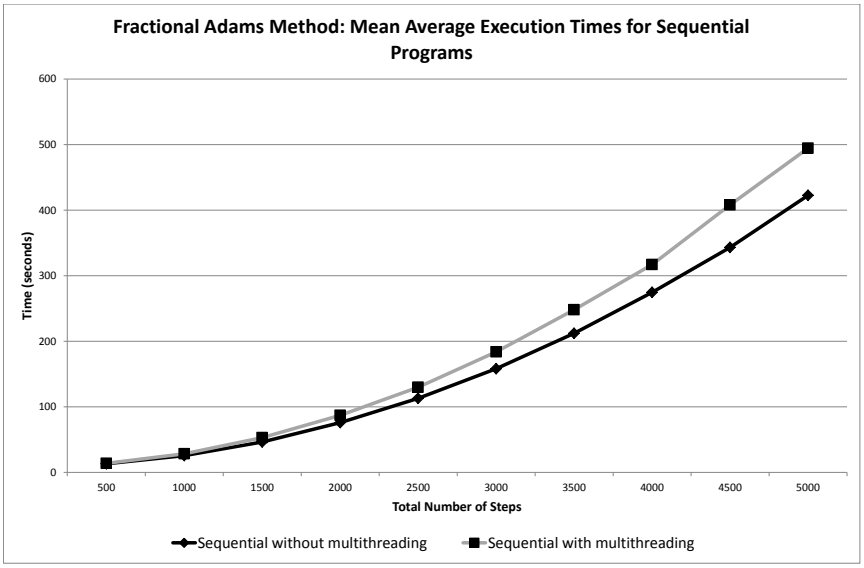

Fig. 9.4: Fractional Adams Method - Execution times of sequential programs with/without multithreading

the data from the implicit multithreaded program 'ABMegDSeqv3' is compared to that of the sequential program 'ABMegDSeqTHREADSv2'. This latter program includes the SPMD command which controls the multithreading capability to a single thread. Figure 9.4 provides the average execution times for various step sizes. Figure 9.5 provides the ratio of sequential programs with multithreading versus the non-multithreaded counterpart. Unlike previous multithreaded programs, the multithreading has had a negative impact upon performance. Figure 9.4 demonstrates an improvement over larger numbers of steps when controls are placed in the program. Viewing the execution time as a ratio in Figure 9.5 indicates the consistency between the two sequential programs.

Although the multithreading data presented in this Chapter presents a negative impact upon performance, it does demonstrate the unpredictability of the resultant program. This unpredictability continues to justify the controlling of multithreading in the remaining prototype programs in Chapter 10.

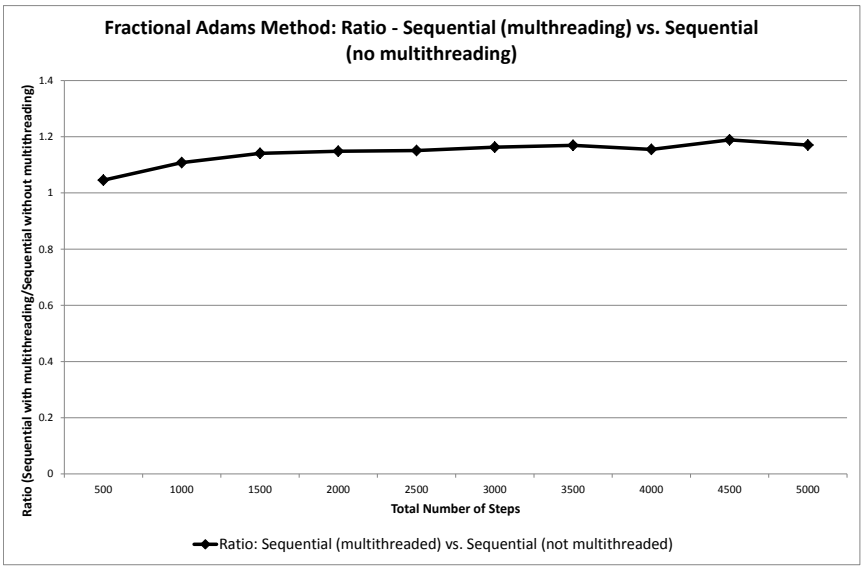

Fig. 9.5: Fractional Adams Method - Ratio of execution times for sequential programs with/without multithreading

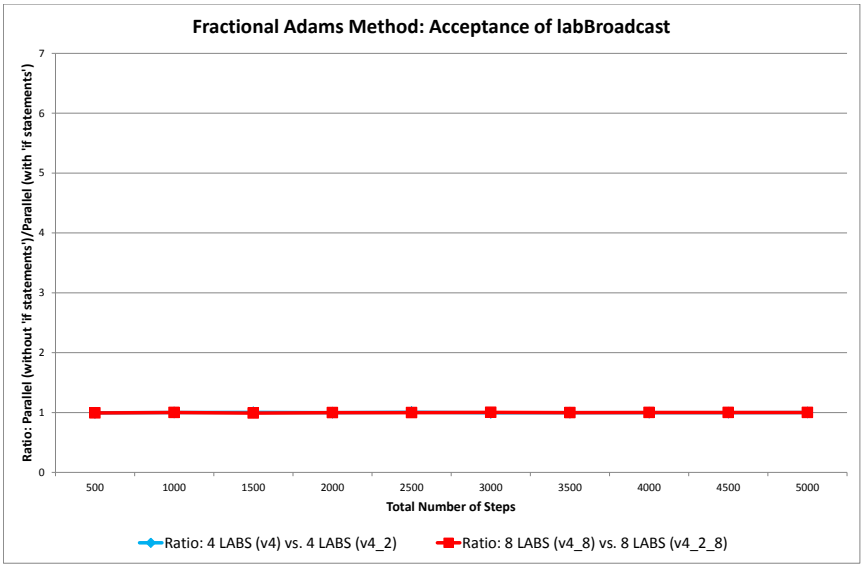

Fig. 9.6: Fractional Adams Method (Acceptance of LabBroadcast) - Ratio of (Mean) Average Execution Times

9.4.2 Acceptance of labBroadcast - Idle Workers

As discussed earlier in Section 9.3 a set of programs was created which included 'if statements'. These 'if statements' allow workers which are not calculating a step in the block ('idle workers') to accept communication from 'active workers'. The new programs to include 'if statements' were:

- ABMegDv4_2 - Four worker program.
- ABMegDv4_2_8 - Eight worker program.

Figure 9.6 provides the ratio of the average execution times for ABMegDv4 and ABMegDv4_8 against the newly created programs (ABMegdv4_2, ABMegDv4_2_8). The raw data used to calculate the ratio is also provided in Table 9.1. The ratio indicates that no positive or negative impact is observed through the use of programs containing the 'if statements' used to accept labBroadcasts.

No. of steps	Sequential (with)	Sequential (without)	4 LABs (v4)	4 LABs (v4_2)	8 LABs (v4_8)	8 LABs (v4_2_8)
500	13.811	13.209	10.930	10.965	12.245	12.314
1000	28.495	25.719	14.169	14.166	13.977	13.959
1500	52.929	46.394	19.504	19.461	16.617	16.759
2000	87.163	75.902	26.873	26.909	20.440	20.476
2500	129.971	112.943	36.481	36.314	25.293	25.337
3000	183.828	158.102	47.932	48.120	31.170	31.118
3500	248.128	212.180	61.472	61.531	38.191	38.285
4000	317.025	274.554	77.358	77.509	46.133	46.106
4500	408.006	343.196	95.169	95.336	55.282	55.293
5000	494.510	422.591	115.203	115.144	65.326	65.260

Tab. 9.1: Fractional Adams Method - (Mean) Average Execution Times (seconds)

9.4.3 Fractional Adams Method - Data Table

For completeness, Table 9.1 provides the mean average execution times for all programs contained in this Section. Averages are rounded to three decimal places. Data is recorded in seconds.

9.5 Theoretical vs Practical Calculations

Table 9.2 summarises the results of the theoretical versus practical benefit of parallel implementation compared to the sequential equivalent. The programs used in these calculations were:

- ABMegDSeqTHREADSv2
- ABMegDv4
- ABMegDv4_8

The example FDE used within these prototype programs originated from the paper by Diethelm [14]. The exact solution contains a Mittag-Leffler function which, when calculated using MatLab (see for example [54]) will also result in an approximation to the exact solution. As a consequence the "exact" solution resulted in an almost constant difference to the approximate data obtained through the Fractional Adams Method. In order to produce the theoretical/practical benefit calculation two step sizes were chosen which would result in a large number of steps. The average execution time for the chosen step sizes ($\frac{1}{900}$ and $\frac{1}{1000}$, at the termination point = 5) were reported

Program	No. of Steps	No. of actions	Time (sec)
Sequential	4500	820,617,784	343.196
Parallel (4 LABs)	4500	93,913,896 (93,845,271)	95.169
Parallel (8 LABs)	4500	36,909,151 (36,788,566)	55.282
Sequential	5000	1,013,047,534	422.591
Parallel (4 LABs)	5000	115,911,271 (115,835,021)	115.203
Parallel (8 LABs)	5000	45,508,146 (45,445,646)	65.326

Tab. 9.2: Theoretical vs. Practical Benefit Calculations

in Section 9.4. This average execution data will now be used in Table 9.2 for practical benefit calculations.

Table 9.2 provides the theoretical/practical benefit calculations for the two problems under consideration. The 'Number of actions' column in Table 9.2 follows the method of Section 4.6.2. An Excel spreadsheet was devised to calculate the number of actions for different step sizes/number of steps for the ABMegDSeqTHREADSv2, ABMegDv4 and ABMegDv4_8 programs. For the parallel programs the result in the Table refers to the LAB with the most number of actions to complete with the number in brackets for the LAB with the least amount of actions.

Total Number of Steps 4500

From Table 9.2 it can be ascertained that the theoretical benefit of parallel implementation is an 88.6% and 95.6% decrease in actions (four-LAB program and eight-LAB program respectively). Practically, the use of the four-LAB program resulted in a 72.3% decrease in execution time. The same calculation for the eight-LAB program resulted in a 83.9% decrease in execution time.

Total Number of Steps 5000

From Table 9.2 it can be ascertained that the theoretical benefit of parallel implementation is an 88.6% and 95.6% decrease in actions (four-LAB program and eight-LAB program respectively). Practically, the use of the four-LAB program resulted in a 72.7% decrease in execution time. The same calculation for the eight-LAB program resulted in a 84.5% decrease in execution time.

9.5.1 FLOPS

Following Section 4.6.3 an assessment of the workload undertaken by the sequential and parallel programs can be made through the calculation of floating point operations. In the Fractional Adams Method programs, the order of arithmetic was $O(N^2)$ where N was the total number of steps in the method.

9.5.2 Comparison of MPCT Results to C++/MPI

Diethelm [14] provides the execution time for the C++/MPI parallel programs to operate for 1-8 processing cores. These execution times are for total number of steps 2.10^5 and 10^6.

Diethelm executes the parallel programs on two Intel Xeon Gainestown quad core processors. He indicates that the clock rate is 3.2GHz. Based upon this description of the servers used in Diethelm's research, it appears he had access to processors which were hyperthread enabled (see [38]).

The execution time for a shared memory architecture was more favourable than the distributed memory architecture due to the passing of data between workers. The execution time for eight workers for step size 2.10^5 was 4.29 seconds (shared memory) and 13.37 seconds (distributed memory).

Replicating Deithelm's research using the MPCT program ABMegDv4_8 on Abel (see Section 6.2) should have resulted in a quick result. In fact the execution time for Abel with total number of steps 2.10^5 was so slow that the program had not completed after five minutes. Comparison of performance between the architectures would not have been favourable given the increased clock speed of Diethelm's architecture and potential application of implicit multithreading. Given the poor performance of the MPCT program and Abel architecture the C++/MPI set-up would be preferable. This performance comparison re-emphasises MPCT as a prototyping tool rather than the final program in a project.

9.6 Discussion Points

Following the research contained in this Chapter it can be observed that the parallel implementation of the Fractional Adams Method (FAM) has resulted

in a decrease in execution time. The theorectical results demonstrated a potential gain from the parallel implementation of the FAM which did translate into a reduction in execution time when compared to the sequential equivalent. Unsurprisingly the theoretical benefit calculations were more optimistic than the practical benefit. When considering a problem with 5000 steps, the difference between the theoretical and practical benefit calculation was 15.9% and 11.1% for the four-LAB program and eight-LAB program respectively.

To conclude this Chapter, the following items will be taken forward in the research:

- The 'block' method continues to be beneficial for structuring parallel programs.
- The inclusion of 'if statements' within a parallel program to accept labBroadcasts on idle workers only resulted in marginal differences to execution time.
- The unpredictability of multithreading in sequential program performance.

10. LUBICH'S FRACTIONAL MULTISTEP METHOD

10.1 Introduction

Lubich's Fractional Multistep Method (FMM) has been outlined in Section 3.4.1. Continuing the work with MatLab Parallel Computing Toolbox (MPCT) this Chapter provides analysis of implemented parallel programs for Lubich's FMM. The following objectives will be satisfied within this Chapter:

- To illustrate how Lubich's FMM can be implemented in MPCT utilising a sequential, a four-worker, and eight-worker parallel architecture.
- To outline the impact of restricting multithreading during the initial stages of a prototype program.
- To continue the assessment of multithreading and its impact upon sequential program performance.
- To assess the performance of the parallel programs resulting from Lubich's FMM implementation in MPCT

10.2 Lubich's FMM: Reminder

Following the work in Section 3.4.1 Lubich's Fractional Multistep Method (FMM) is defined in Equation 3.79 and repeated here for the Reader:

Lubich's FMM Algorithm

$$y_n = y_0 + h^\alpha \sum_{j=0}^{n} \omega_{n-j} f\left(jh, y_j\right) + h^\alpha \sum_{j=0}^{s} w_{nj} f\left(jh, y_j\right)$$

where:

- α is the order of the fractional derivative.
- y_n is the calculated step under consideration.

- h is the step size and n is the total number of steps required to reach the terminal value t. The total period under consideration can therefore be expressed as $t = nh$. (In this research we only consider equispaced steps).

- w_{nj} are the starting weights.

- ω_{nj} are the convolution weights.

- s represents the end of the starting phase. The starting phase will require the initial value y_0, and predicted values for the equation at points y_1 to y_s.

- y_n requires prediction which could be through another numerical method such as the Newton's Method [15] or other Multistep Method. (In the research proceeding the Fractional Adams Bashforth Method was used as a predictor).

10.2.1 Calculation of Weights

As discussed in Section 3.4.1 the convolution weights ω_{n-j} are obtained through the Taylor's Series Expansion to the α^{th} power of the function $\omega(z)$. This function is calculated using the characteristic polynomials of a chosen Linear Multistep Method (the example used in this Chapter utilises the Trapezium Rule). The Starting Weights w_{nj} are determined from a linear system of equations. The linear system is obtained using the generating function $g(t) = t^{\gamma}$ where $\gamma \in A$ and $A = \left\{ \gamma = j + l\alpha : j, l \in \{0, 1, \dots\}, \gamma \leq p - 1 \right\}$. The resultant linear system for the starting weights is ill-conditioned and therefore methods such as GMRES should be employed.

10.3 Lubich's Fractional Multistep Method: Implementation in MatLab

Following the same scheme as Chapters 8 and 9 a 'block' approach was utilised (see Section 5.3.3). Prior to the block implementation, an **initial section** of code was implemented outside of the parallel programming construct. Prior to starting the main algorithm section, this initial section provided all workers with a common set of variables, e.g. weights. As with previous MPCT implementations, the SPMD command was included to allow each worker to process its own step size whilst using the same program. The

SPMD command was also needed to restrict multithreading in the sequential program (see Section 6.6). During each block, workers communicated results between each other, this was achieved through the labBroadcast and labSend/labReceive commands (Section 5.4).

The initial section described in the previous paragraph appeared to be sequential code. However, later research discovered this section was implementing implicit multithreading. Although this implicit multithreading in the initial section was a continuation of previous MPCT prototype program structures, the amount of work undertaken in Lubich's FMM programs warranted additional investigation. As a consequence the 'hybrid' programs listed below follow the standard procedure used within this research. Additionally the 'restricted threading' programs included the 'maxNumCompThreads' command (Section 6.6) to undertake the initial section in a truely sequential (single) worker fashion.

Figure 10.1 provides an annotated version of a 'restricted threads' program written in a simplified pseudocode. A fully annotated version of the program 'LubichParallel8Labsv4INLOOPAttemptToControlThreads', upon which Figure 10.1 is based, is provided in the Appendix D. As described, this program restricts to one thread of operation in the initial section. The main algorithm section begins when the MatLab Pool is opened to the eight-workers and the SPMD command is called.

The programs created for Lubich's FMM experimentation are provided in the list following and are included on the CD attached to this thesis.

- **LubichParallelSeqINLOOP** - A sequential program for Lubich's FMM. A MatLab Pool is called for one worker however multithreading is still in operation. An overarching loop repeats the program thirteen times, displaying the execution time after each iteration for averaging purposes (see Section 6.3).

- **LubichParallelSeqINLOOPThreads** - 'Hybrid' program. Same structure as 'LubichParallelSeqINLOOP' program with the addition of a SPMD statement to restrict to only one worker at the point where the parallel programs would be operating with multiple workers. Again this program has an overarching loop which repeats the execution of the program thirteen times, displaying the execution time after each iteration for averaging purposes (see Section 6.3).

```
Termination point set
Step size h defined by user
Value of fractional derivative set
Initial starting phase 's' set
For loop 1 to 13
  Restrict to one thread (maxNumCompThreads)
  Start timing the program
  Create x array and assign initial condition
  Create array 'startcoef' for coefficients used with predictor
  Calculate the predictor coefficients and store in 'startcoef'
  Create arrays for calculation of convolution weights
  Calculate the convolution weights by multiplying elements of the  arrays together
  Remove restriction on threads
  Open the parallel pool
  SPMD
    for loop  - puts the steps into blocks of equal size to workers in pool
      Calculation of starting weights
      LAB1 calculates first value of block
      Communication to other LABS of result of LAB1
      Other LABS wait for previous result and calculate their step
      Once calculated approximation, send on to the  LABS in front
    end
  end
  Close the pool of LABS
  Stop timing and display result
End of loop for repetition of algorithm
```

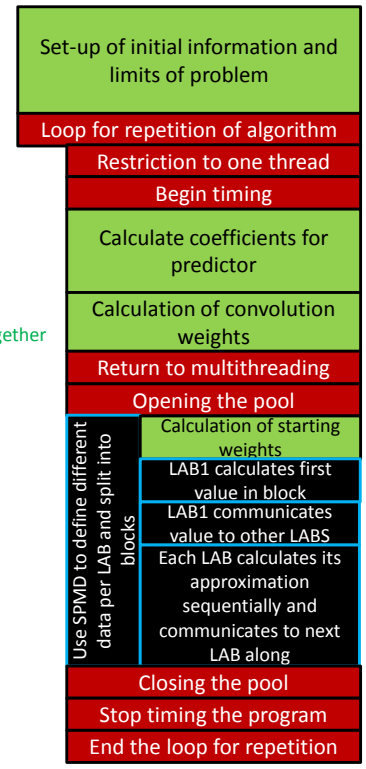

Fig. 10.1: Lubich's FMM: Simplified program stucture

- **LubichParallelSeqINLOOPAttemptToControlThreads** - 'Restricted threading' program. Third sequential program which builds on 'LubichParallelSeqINLOOPThreads'. The program has an additional 'maxNumCompThreads' command to restrict operation to a single thread during the initial stage. Again this program has an overarching loop which repeats the execution of the program thirteen times, displaying the execution time after each iteration for averaging purposes (see Section 6.3).

- **LubichParallel4Labsv4INLOOP** - 'Hybrid' program. Parallel version of Lubich's FMM utilising four workers in the MatLabPool and multithreading in the initial phase. Again this program has an overarching loop which repeats the execution of the program thirteen times, displaying the execution time after each iteration for averaging purposes (see Section 6.3).

- **LubichParallel4Labsv4INLOOPAttemptToControlThreads** - 'Restricted threading' program. Parallel version of Lubich's FMM utilising four workers. The program has the same structure as 'LubichParallel4Labsv4INLOOP' with the addition of the 'maxNumCompThreads' command in the initial section to restrict to a single thread of operation. Again this program has an overarching loop which repeats the execution of the program thirteen times, displaying the execution time after each iteration for averaging purposes (see Section 6.3).

- **LubichParallel8Labsv4INLOOP** - 'Hybrid' program. Parallel version of Lubich's FMM utilising eight workers in the MatLabPool and multithreading in the initial phase. Again this program has an overarching loop which repeats the execution of the program thirteen times, displaying the execution time after each iteration for averaging purposes (see Section 6.3).

- **LubichParallel8Labsv4INLOOPAttemptToControlThreads** - 'Restricted threading' program. Parallel version of Lubich's FMM utilising eight workers. The program has the same structure as 'LubichParallel8Labsv4INLOOP' with the addition of the 'maxNumCompThreads' command in the initial section to restrict to one thread of operation. Again this program has an overarching loop which repeats the execution of the program thirteen times, displaying the execution time after each iteration for averaging purposes (see Section 6.3).

To prevent hardware limitations impacting upon performance all program

results were performed upon Abel (see Section 6.2.1).

In order to ascertain if performance was enhanced by the parallel implementation the step size h was varied. Decreasing the step size resulted in an increased total number of steps (reported in the Figures which follow). Comparisons between the execution time of the four, eight and sequential versions of the programs were undertaken at each step size. The value of the step size h was the only variable, all other inputs were maintained and were:

- Termination point $T = 1$
- Value of fractional derivative $\alpha = 0.25$
- The Trapezium Rule was used as the underlying LMM which has order $p = 2$
- The initial condition $y_0 = 0$

The Fractional Adams Bashforth Method (FABM) was used to calculate the starting phase results and to predict y_n. The use of the FABM was a variation on the typical approach where Newton's Method is used as the predictor (see Diethelm et al [15]) .

10.3.1 Results from MatLab

As discussed in Section 10.3 two sets of programs were created. 'Hybrid' programs refer to the multithreaded initial section on a program before the use of parallelism in the main algorithm section. The 'Restricted threading' programs included the use of the 'maxNumCompThreads' command to remove multithreading in the initial section. The restriction to threading was removed prior to opening the MatLab Pool.

Hybrid Programs

Below are the programs used for analysing performance of prototype programs where a 'hybrid' approach was utilised:

- LubichParallelSeqINLOOPThreads
- LubichParallel4Labsv4INLOOP

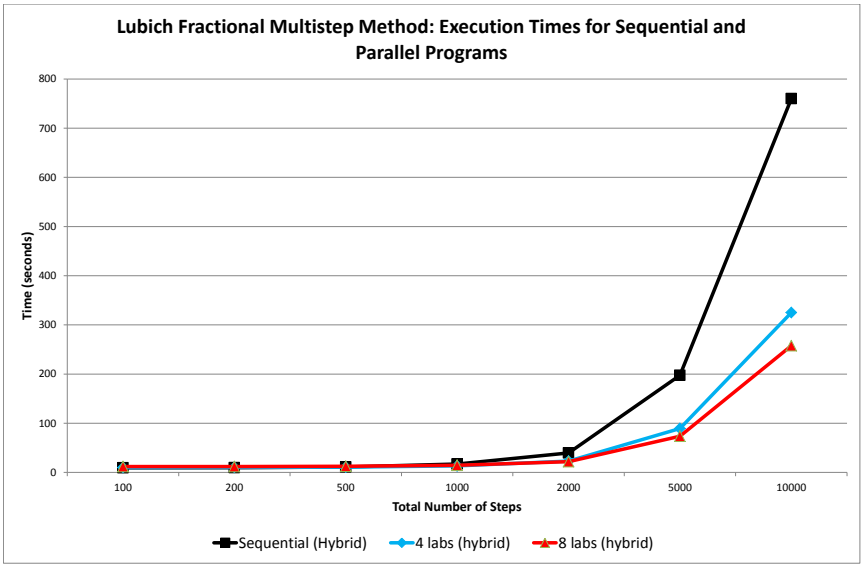

Fig. 10.2: Lubich's FMM Hybrid: Execution Times for sequential and parallel programs

- LubichParallel8Labsv4INLOOP

The (mean) average execution times for the different programs which utilised the 'hybrid' approach are provided in Figure 10.2. In Figure 10.3 the average execution time has been expressed as a ratio based upon the Speed up Equation 4.1 (see Section 4.6.1). The Speed up equation, repeated for the Reader is Speedup $(n) = \frac{T(1)}{T(n)}$. The exact runtime averages are provided in Table 10.1.

Figure 10.2 illustrates the improved performance of the parallel programs when compared to the sequential counterpart over an increased number of steps. By 10,000 steps the eight-LAB parallel program has an execution time which is a 66.1% decrease upon the sequential code. Considering Figure 10.3 the 'Tipping Point' is visible around 500-1000 steps for the parallel programs. Neither parallel implementation is scalable. However the eight-LAB program does almost achieve a scale-up of three against its sequential counterpart. The eight-LAB program also achieves a better performance than the four-LAB parallel prototype at approximately 1500 steps.

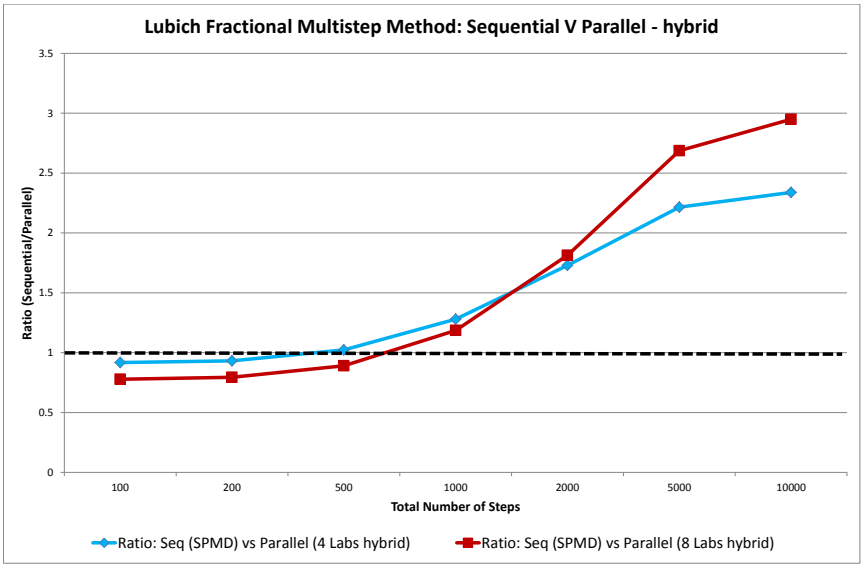

Fig. 10.3: Lubich's FMM Hybrid: Speed ratio (sequential/parallel)

Restricted Threading

Furthering the research into the impact of multithreading, Table 10.2 and Figure 10.4 provide the (mean) average execution times for the 'restricted threading' programs. The programs used in this analysis are:

- LubichParallelSeqINLOOPAttemptToControlThreads
- LubichParallel4Labsv4INLOOPAttemptToControlThreads
- LubichParallel8Labsv4INLOOPAttemptToControlThreads

In Figure 10.4 the difference between the four-LAB parallel and sequential programs is 450 seconds. This represents a 58% decrease in execution time when using the parallel program. Figure 10.5 provides the ratio of Sequential to Parallel program execution speed. These calculations were based upon the Speed-up Equation 4.1.

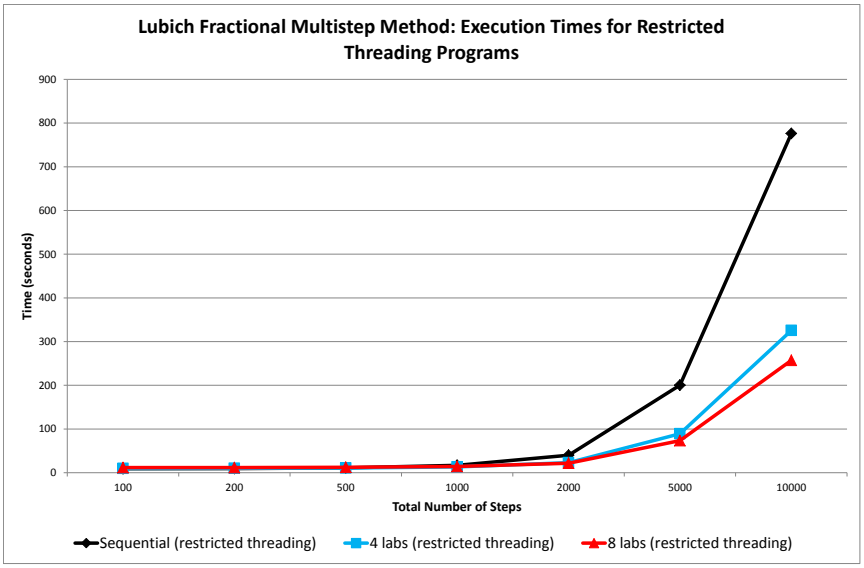

Fig. 10.4: Lubich's FMM Restricted Threading: Execution Times for Sequential and Parallel Programs

No. of steps	Sequential	Parallel (4 LABs)	Parallel (8 LABs)
10	9.128	10.026	11.866
20	9.119	10.027	11.858
50	9.136	10.027	11.755
100	9.213	10.047	11.848
200	9.489	10.181	11.952
500	11.135	10.887	12.500
1000	17.081	13.344	14.399
2000	39.627	22.926	21.853
5000	197.591	89.172	73.545
10000	760.224	325.098	257.812

Tab. 10.1: Runtime of Hybrid Lubich's FMM Programs (time in seconds)

Figure 10.5 demonstrates the 'Tipping Point' at which the parallel programs perform quicker than the sequential equivalent. This appears to occur around 500 steps for the four-LAB program and between 500-1000 steps for the eight-LAB program. These results mirror that observed in Figure 10.3.

Table 10.2 provides the exact runtime averages for the sequential and parallel programs with restricted threading.

No. of steps	Sequential	Parallel (4 LABs)	Parallel (8 LABs)
10	9.126	9.961	11.755
20	9.119	9.957	11.791
50	9.148	9.987	11.790
100	9.205	9.998	11.765
200	9.458	10.107	11.924
500	11.141	10.830	12.457
1000	16.936	13.294	14.365
2000	39.991	22.892	21.776
5000	200.259	89.246	73.388
10000	775.991	325.606	257.020

Tab. 10.2: Runtime of Lubich's FMM Programs with restricted threading (time in seconds)

It is observed in Table 10.2 that the runtime of the eight-LAB program is a 66.9% decrease on the sequential counterpart (10,000 steps) - a similar result to the hybrid programs in Table 10.1.

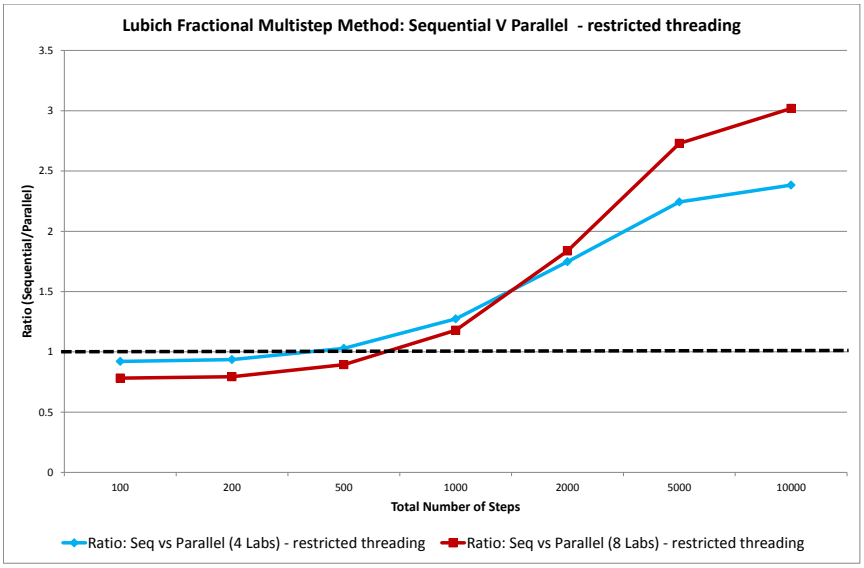

Fig. 10.5: Lubich's FMM Restricted Threading: Speed ratio (sequential/parallel)

Figure 10.6 provides the ratio of 'Restricted Threading' to 'Hybrid' for the sequential, four-LAB and eight-LAB programs. Very little difference with respect to execution time was observed.

Sequential Programs

Figure 10.7 provides the execution times for three versions of sequential programs. The programs used in this analysis were:

- Sequential (restricted threading) - This program operates a restriction to one thread in the initial section. An SPMD command is used to ensure only one LAB is in operation during the main algorithm section. This is the program 'LubichParallelSeqINLOOPAttemtToControlThreads'.
- Sequential (Hybrid) - This is the program 'LubichParallelSeqINLOOPThreads'. Multithreading is present in the initial section and SPMD is used to restrict the algorithm phase to just one worker.

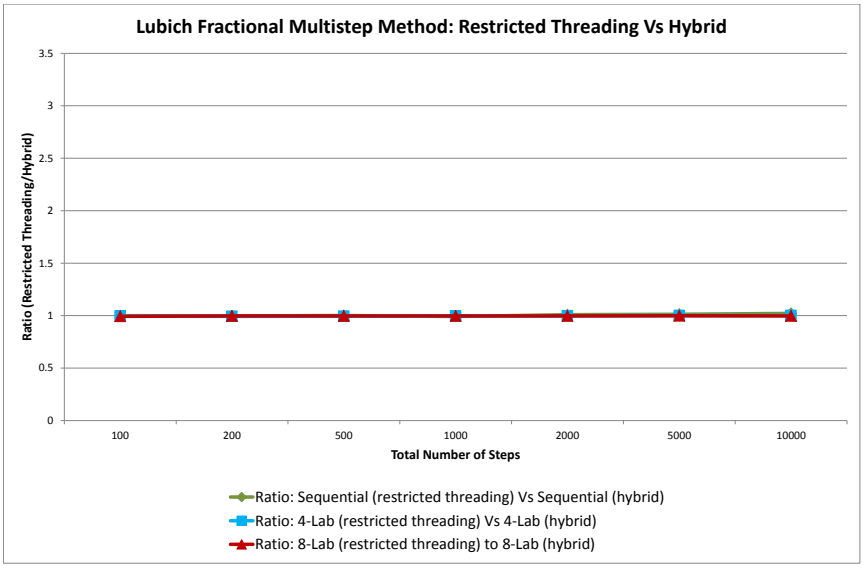

Fig. 10.6: Lubich's FMM: Restricted Threading vs Hybrid

- Sequential (multithreading enabled througout) - This program does open a pool for one-LAB. However multithreading is in operation throughout. This is the program 'LubichParallelSeqINLOOP'.

Figure 10.7 demonstrates the differences between the three sequential programs. The 'Hybrid' and 'Restricted Threading' programs produce similar results. However the impact of multithreading is evident when comparing the Sequential (multithreading enabled throughout) data against the other sequential programs. By 10,000 steps the sequential programs without/restricted multithreading produced a 93.9% increase in execution time when compared to the multithread-enabled program.

10.4 Theoretical vs. Practical Benefit Calculations

Table 10.3 summarises the results of the theoretical versus practical benefits of parallel (hybrid) implementation compared to the sequential (hybrid) equivalent. The example FDE used in the MPCT programs was provided in the paper by Diethelm et al [15], for which an exact solution is known.

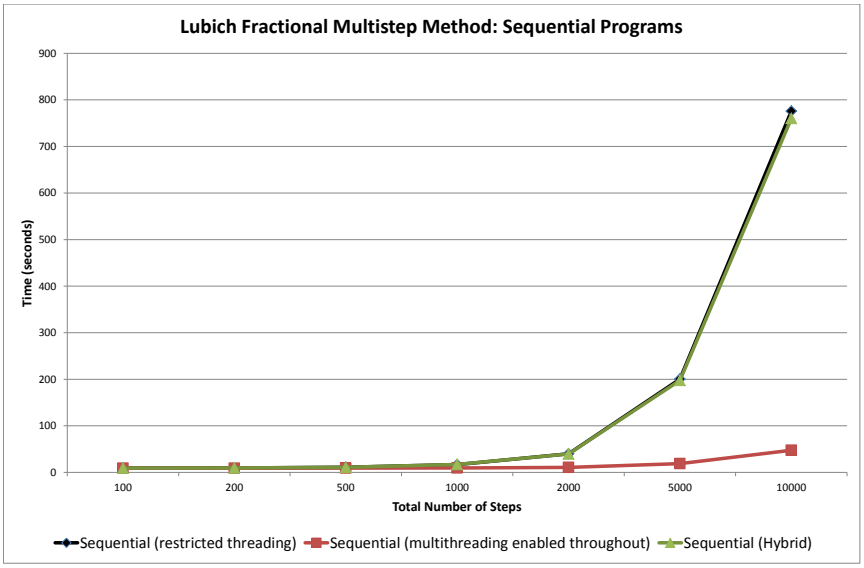

Fig. 10.7: Lubich's FMM: Sequential Program Runtime (seconds)

Based upon the desired level of accuracy indicated in the table, the total number of steps ('No. of steps') was determined by comparing the exact and approximated results for the example FDE. (The termination point remained the same). To quicken this determination one of the MPCT programs was altered to include a while loop which would continue to operate until the desired level of accuracy was achieved.

Program	Error tolerance	No. of steps	No. of actions	Time
Sequential	$10^{(-3)}$	320	6,133,412	9.928
Parallel (4 LABs)	$10^{(-3)}$	320	1,743,115	10.352
Parallel (8 LABs)	$10^{(-3)}$	320	1,332,040	12.139
Sequential	$10^{(-4)}$	2560	381,667,172	58.801
Parallel (4 LABs)	$10^{(-4)}$	2560	106,412,715	30.749
Parallel (8 LABs)	$10^{(-4)}$	2560	81,261,960	28.109

Tab. 10.3: Theoretical vs. Practical Benefit Calculations

Looking at the data in Table 10.3 and considering the 10^{-3} level of accuracy; theoretically the benefit of using the four-Lab and eight-Lab programs

compared to the sequential equivalence results in a decrease of actions, 71.6% decrease for the four-worker program and 78.3% decrease for the eight-worker program. Considering the practical benefit for this desired level of accuracy, the four-LAB parallel program was 4.3% slower than the sequential program. The eight-Lab program obtained 22.3% increase in execution time compared to the sequential program.

Returning to Table 10.3 and considering the 10^{-4} level of accuracy was more satisfying. Theoretically the four-LAB and eight-LAB parallel programs results in a decrease in actions of 72.1% and 78.7% respectively. Considering the practical benefit for this desired level of accuracy the four-LAB parallel program obtained a 47.7% decrease in execution time when compared to the sequential program. A more impressive result was observed with the eight-LAB program which was a 52.2% decrease in execution time compared to the sequential equivalent.

These two theoretical/practical benefit calculations demonstrate the impact of other factors upon the execution time of a program. Although theoretically both problems could have benefitted over 70% by implementing in parallel, this does not seem proportional given the doubling of the MatLab Pool to eight workers. In both the practical and theoretical calculations the impact of other overheads such as communication can be observed. The theoretical benefit calculations include LAB comunications. The increased number of workers results in an increased number of communciation points which grows as the problem size increases. This is also the case with the practical benefit calculations and is particularly observable with the smaller size problem of 320 steps. The practical benefit calculations will also demonstrate the impact of waiting for information from other LABs in the pool which would not be measured in the theoretical results. By increasing the problem size to 2560 steps, the other overheads were reduced and a positive result observed.

10.4.1 FLOPS

Following the information presented in Section 4.6.3 a calculation of the floating point operations has been undertaken for the sequential and parallel prototype programs. The sequential and parallel programs resulted in a FLOP calculation of $O\left(n^2\right)$.

The additional benefit of the spread sheet devised for the theoretical ben-

efit calculations is guiding the user to the most expensive computational line. In other numerical methods researched in this project the FLOP and spread sheet have highlighted the same line of code for computational expense. This is not the case with Lubich's FMM where the theoretical spread sheet for the sequential program highlighted a different line of code. This line of code had multiple calls for information from memory and resulted in the increased number of actions which would not be reflected in the FLOP calculations.

10.5 Alternative MatLab Implementation

The MatLab programs above represent a simple, self-contained implementation of Lubich's FMM. In Baleanu et al [4] Lubich's FMM has been implemented in a different structure using MatLab across three programs. In this structure the value of the fractional derivative α can be varied. The first program provides the starting values for the fractional differential equation utilising a simple Newton's Method. The second program constructs a matrix for the weights of the algorithm providing the user with a choice of methods for determining the starting weights. This second program has a convolution weights matrix obtained via Automatic Differentiation. The third program computes the approximate solutions.

This alternative implementation of Lubich's FMM provides the user more input into how the algorithm is implemented. The overall structure of the implementation in Baleanu et al [4] does not utilise MPCT. The work in this thesis is a step in furthering knowledge towards the development of prototype parallel programs. Hence information gained from this project could help to adapt the alternative programmatical structures described in Baleanu et al [4] in the MPCT environment.

10.6 Discussion Points

The research in this Chapter has evidenced that time-efficiencies can be gained through the implementation of Lubich's FMM in MPCT. The data presented in Section 10.3.1 presented the 'Tipping Point' between 500 and 1000 steps.

The impact of multithreading was further explored with the creation of the 'Restricted Threading' programs. The differences between the 'hybrid'

and 'restricted threading' programs produced no diversity in data. However the impact of restricting multithreading upon sequential code was significant. Comparisons between the sequential program results demonstrated a 93.9% increase in execution time when compared to the multithreaded equivalence.

Following the research of this Chapter the following items will be taken forward for further consideration:

- The created parallel programs have yet again demonstrated a positive impact over large-sized problems.
- Restricting of multithreading in the preparatory section of a program does not yield a difference when compared to the standard 'hybrid' program structure.
- Multithreading in the sequential program proved very efficient in the execution of Lubich's FMM.
- The overheads such as communication have had a significant impact upon performance. Considering the communication overheads, would a communication strategy where data is only sent to the next LAB in the pool produce a significant result?

11. CONCLUSION

In this thesis we have successfully demonstrated that MatLab Parallel Computing Toolbox (MPCT) can be used to create parallel programs which provide speedup to a sequential counter-part. However, the successes of prototyping numerical methods have also highlighted the need to understand inbuilt mechanics of MPCT, in particular the use of multithreading in matrix and element-wise operations and its impact upon execution time. Early research into MPCT found sequential programs performing incredibly quickly despite the attempt to open/close a MatLab Pool with only one worker. Further investigation discovered the implicit multithreading was 'ignoring' the call for single-worker operation which mirrors other design features of MPCT. For example where no Pool is in operation the parfor command is 'ignored' and converted to a 'for-loop' instead, see [58]. Data in Chapter 9 demonstrated the opposite situation, that multithreading was a detriment to the performance of sequential programs. In either situation it is clear that multithreading needs to be recognised and controlled, therefore providing a fair comparator to parallel implementations.

Although performance gains in using parallel programs have been possible, scalability has remained illusive in the prototype programs within this thesis. To understand why, users of MPCT not only need to know the general barriers to scalability but more particularly the specific origins of the software as a tool to exploit embarassingly parallel problems where little communication between workers was required. This thesis has explored the ability of MPCT to communicate between workers in a fine-grained problem across a small scale architecture. The research presented in Chapter 6 demonstrated the basic communication commands available to a programmer, in particular the slight improvement in execution time when the labSend/labReceive command was used in conjunction with labBroadcast. Chapter 6 also provided data on time taken for the MatLab Pool to open/close which ranges from 8.876 seconds for one-worker to 11.372 seconds for eight-workers which has to be accounted for when assessing viability of the prototype parallel programs.

Initial attempts at prototyping parallel programs concentrated on nu-

merical methods for Ordinary Differential Equations (ODEs). These initial parallel programs took advantage of the MatLab Pool size to implement an Adapted Richardson Extrapolation Scheme. Results for the Runge-Kutta Method were successful but the poor load-balance incurred through choice of step size resulted in poor scalability. This area could be further explored through the better selection of step sizes which are closer in size and further adaptation of the Richardson Extrapolation Scheme to exploit the calculated approximations.

Following the creation of prototype parallel programs for ODEs, research continued to consider cases where the derivative was fractional. Alterations to the structure of the prototype parallel programs were explored. Most significantly, and mirroring the work of Diethelm in [14] a 'block' approach with SPMD command was found to be more favourable in creating a good load-balance with the problems investigated. A slight change in the Fractional Adams Method parallel programs to accept data on 'idle' workers yielded little changes in execution times. The creation of hybrid programs for Lubich's Fractional Multistep Method (FMM) which were compared to programs where threading was restricted also yielded little change in execution time.

Throughout the research in this thesis theoretical benefit of implementation in parallel was calculated. This concept of 'Number of Actions' through recording the fetching, storing, calculating of data within each line of code can be used to identify whether parallel implementation can be beneficial and whether load-balance is suitable. This aspect was particularly highlighted with the theoretical calculations in Chapter 7 where the poor-load balance between the workers was evident.

As described above, several attempts have been made in the course of this research to enhance execution time of the resultant parallel programs. These ideas are summarised below as avenues which have yielded unsatisfactory results:

- The appropriate use of labSend/labReceive with labBroadcast can yield minor improvements to execution time of a parallel program compared to the exclusive use of labBroadcast.
- The inclusion of 'if-statements' to allow the acceptance of data on 'idle' workers within a Pool does not yield a quicker execution time.
- Restricting threading on sequential elements of parallel programs yielded

little change in execution time when compared to un-restricted threading.

11.1 Key Principles

In summary, this research has provided the following key principles for others who wish to use MPCT as a prototyping language for problems which lead to fine-grained parallel programs:

1. To gain fair sequential comparitors, restriction of multithreading should be considered. Restrictions can be put in operation through opening a Pool with only one worker and using an SPMD command.

2. The 'block' method with SPMD proves useful in structuring parallel programs to create a good load-balance.

3. Consideration given to the number of 'actions' a program undertakes can provide a good indication of load-balance. Where the number of 'actions' indicate a good load-balance and execution time is poor, this method can also indicate additional communication overheads or lack of complexity in the given problem.

11.2 Ideas for Future Research

In the course of this research other ideas have emerged as potential areas of investigation. These ideas are detailed below:

- The adoptation of the 'gmres' function in MatLab. This function could be used to solve the ill-conditioned Vandermonde Matrix which is created when calculating the starting weights of Lubich's Fractional Multistep Method.

- The use of a multistage method within Lubich's FMM to provide a Fractional Multivalue Method. - Would this provide a more accurate approximation? In Baleanu et al [4] the authors have indicated that the accuracy of the initial approximations have little effect upon the convergence of the resultant method. However what would be the effect of including off-step approximations?

- Impact of using a Beowulf Cluster - Returning to the original design brief of MPCT, would moving to a cluster be more beneficial? Would workers on a small cluster across a closed network mean that each

worker could implement implicit multithreading on their aspect of the program? Could the programs be designed to exploit this alternative environment? For example: the use of the cluster for different step sizes to later apply Richardson Extrapolation?

- The adoption of a communication strategy where data is only passed to the next worker in the pool. How does this strategy impact upon the execution times of the resultant parallel programs?

- The further adaptation of the Richardson Extrapolation Scheme - As indicated earlier, the Richardson Extrapolation Scheme could be further adapted to accept four approximations which have step sizes very close to each other. For example: LAB1 could consider 0.1, LAB2 could consider 0.09, LAB3 could consider 0.08, and LAB 4 could consider 0.07. By choosing step sizes close to each other the load-balance issue would be reduced.

APPENDIX

A. RUNGE-KUTTA METHOD FOR ORDINARY DIFFERENTIAL EQUATIONS - ANNOTATED PROGRAM

Following the work in Chapter 7 an annotated version of the 'ParLambertEx6REVariableLastv2' program is provided below. This program operates across a four-worker pool. The Adapted Richardson Extrapolation (see Section 7.2.2) scheme is utilised, and the extrapolated data is compared to the exact results for the example.

Code	Commentary
%Lambert P40 Example 6	-
hh=input('Enter the step size ');	User to input the step size
Last=input('Enter the last variable');	User to input the last value of time period
%p=The order of the underlying method.	-
tic;	Start stop watch
p=2;	Order of numerical method
matlabpool 4	Opening the MatLab Pool
% ppp=parpool (4);	version)
spmd	Single program multiple instruction
h=hh/(2^(labindex-1));	Step size for LAB
u=zeros(size(0:h:Last));	Array to store 'u' values
v=zeros(size(0:h:Last));	Array to store 'v' values
u(1)=0.5;	Storing initial value for u(1)
v(1)=-3;	Storing initial value for v(1)
for i=2:((Last/h)+1)	elements
tempU=u(i-1)+((h/2)*v(i-1));	Function value of 'u' element for k_1
tempV=v(i-1)+((h/2)*((v(i-1)*(v(i-1)-1))/u(i-1)));	Function value of 'v' element for k_1
k2v=(((tempV)^2)-tempV)/(tempU+((h/2)*tempV)-(h*tempV)+(h/2));	Calculation of 'u' element of k_2
k2u=tempV+((h/2)*k2v);	Calculation of 'v' element of k_2
u(i)=u(i-1)+((h/2)*(v(i-1)+k2u));	Calculation of 'u' element of RK step using k2u and value of 'u' within k_1
tempk1v=(v(i-1)*(v(i-1)-1))/u(i-1);	Function value of 'v'
v(i)=v(i-1)+((h/2)*(tempk1v+k2v));	tempk1v
end	End of for loop
if labindex==2	Inspection of labindex number
u2=u;	Temporary assignment of u data from LAB2 to u2
v2=v;	Temporary assignment of v data from LAB2 to v2
labSend(u2,1);	Send u2 data to LAB1
labSend(v2,1);	Send v2 data to LAB1
elseif labindex==3	Alternative for when labindex is 3
u3=u;	Temporary assignment of u data from LAB3 to u3
v3=v;	Temporary assignment of v data from LAB3 to v3
labSend(u3,1);	Send u3 data to LAB1
labSend(v3,1);	Send v3 data to LAB1
elseif labindex==4	Alternative for when labindex is 4
u4=u;	Temporary assignment of u data from LAB4 to u4
v4=v;	Temporary assignment of v data from LAB4 to v4
labSend(u4,1);	Send u4 data to LAB1
labSend(v4,1);	Send v4 data to LAB1
elseif labindex==1	Alternative for when labindex is 1
u2 = labReceive(2);	Receive data from LAB2 and assign to variable u2
v2 = labReceive(2);	Receive data from LAB2 and assign to variable v2
u3 = labReceive(3);	Receive data from LAB3 and assign to variable u3
v3 = labReceive(3);	Receive data from LAB3 and assign to variable v3
u4 = labReceive(4);	Receive data from LAB4 and assign to variable u4
v4 = labReceive(4);	Receive data from LAB4 and assign to variable v4
x=0:hh:Last;	Create array 'x' for storing norm data
uu=zeros(size(x));	Create array 'uu' for exact values of 'u' element
vv=zeros(size(x));	Create array 'vv' for exact values of 'v' elements

Code	Commentary
uu(1)=(1+(3*(exp(0))))/8;	Assign first value as exact value of 'u' using initial value
vv(1)=(-3)*(exp(-8*(0)));	Assign first value as exact value of 'v' using initial value
x(1)=normest([uu(1);vv(1)]-[u(1);v(1)]);	Calculate the norm of exact and RK value of initial value
uRE = zeros(size(x));	Create array for extrapolated 'u' element
vRE = zeros(size(x));	Create array for extrapolated 'v' element
for j=2:((Last/hh)+1)	For loop for calculating the extrapolated results and comparing to exact data
a = (8*j)-7;	other LABs
b = (4*j)-3;	other LABs
c = (2*j)-1;	other LABs
bottom = (8^p)-(4^p)-(2^p)+1;	Extrapolation Scheme
uRE(j)=(((8^p)*(u4(a)))-((4^p)*(u3(b)))-((2^p)*(u2(c)))+u(j))/bottom;	Extrapolation of the 'u' element using the calculated data received from LABs
vRE(j)=(((8^p)*(v4(a)))-((4^p)*(v3(b)))-((2^p)*(v2(c)))+v(j))/bottom;	Extrapolation of the 'v' element using the calculated data received from LABs
uu(j)=(1+(3*(exp(-8*(hh*(j-1)))))))/8;	Calculation of exact result for 'u' at current step
vv(j)=(-3)*(exp(-8*(hh*(j-1))));	Calculation of exact result for 'v' at current step
x(j)=normest([uu(j);vv(j)]-[uRE(j);vRE(j)]);	Norm of exact and extrapolated result
end	Completed for loop
%disp('The value of x is ...')	
%disp(x);	
end	End of 'if statement'
end	End of SPMD
% delete(ppp)	(Alternative to closing MatLab Pool)
matlabpool close	Close MatLab Pool
toc;	Stop the timer

B. DIETHELM-CHERN - ANNOTATED PROGRAM

Following the work in Chapter 8 the program 'RevisedFractionalProgParallelv2_8LabINLOOP.m' is annotated for the Reader below. This program is for an eight-worker pool and uses a 'block' approach (Section 4.5).

Code	Commentary
%Parallel FDE Program Revised structure 14/07/13	
q=input('Enter the value of the fractional derivate ');	User input of fractional derivative
jay=input('Enter the number of steps within each unit ');	User input of steps within one unit
B=input('Enter the value of beta, the coefficient which needs to be less than or equal to zero');	User input of beta coefficient
initialcon=input('Enter the value of the initial condition ');	User input of initial condition data
Last=input('Enter the last variable ');	User input of `Last' value of period for integration
for cloop=1:13	Loop used to rerun algorithm 13 times
disp('Loop')	Display text
disp(cloop)	Display the loop indicating which program run currently executing
tic	Start stopwatch
jayk=Last/(1/jay);	Calculate the total number of steps needed to complete algorithm
x=zeros((jayk+1),1);	Create array x which is same size as the total number of steps
x(1,1)=initialcon;	Set the initial condition data in as first value of x array
alpha=zeros((jayk+1),(jayk+1));	Create matrix for alpha coefficients
for j=2:(jayk+1)	Loop used to calculate the alpha coefficients
for k=1:j	Loop for calculating the relevant coefficients of alpha
if k==1	If condition for calculating the first alpha coefficient within step
alpha(k,j)=-1/(q*(1-q)*((j-1)^(-q)));	Equation for first alpha coefficient
elseif k==j	Alternative if condition for last alpha coefficient within step
first=(q-1)*((k-1)^(-q));	Element of the last alpha coefficient
second=(k-2)^(1-q);	Element of the last alpha coefficient
third=(k-1)^(1-q);	Element of the last alpha coefficient
alpha(k,j)=(first-second+third)/(q*(1-q)*((j-1)^(-q)));	Equation for last alpha coefficient utilising previously calculated values
else	Alternative if condition for all other alpha coefficients
fourth=2*((k-1)^(1-q));	Element of the alpha coefficient
fifth=(k-2)^(1-q);	Element of the alpha coefficient
sixth=(k)^(1-q);	Element of the alpha coefficient
alpha(k,j)=(fourth-fifth-sixth)/(q*(1-q)*((j-1)^(-q)));	Equation for the alpha coefficient utilising previously calculated values
end	end of if condition
end	end of for loop beginning k=1:j
end	end of for loop beginning j=2:(jayk+1)
matlabpool 8	Open the matlabpool

Code	Commentary
`spmd`	Begin the SPMD command
`for endval=1:ceil((Last*jay)/numlabs)`	Loop used to divide the total number of steps into blocks of equal size to number of labs in the pool
`l1=(numlabs*(endval-1))+labindex;`	Calculates the step current worker is using
`if l1<=(Last*jay)`	If condition to determine if current step is within the steps requiring calculation
`sumI=0;`	Set sumI as zero
`if l1<(numlabs+1)`	If condition to check if current step is in first block
`sumI=0;`	Set sumI as zero
`else`	Alternative to if condition when not in first block
`for hh=2:(l1-labindex+2)`	Loop for calculating the sum element of the algorithm utilising previous values from old blocks
`sumI=sumI+(alpha((l1+3-hh),(l1+1))*(x((hh-1),1)));`	Cumulative sum of previous values from old blocks.
`end`	end of for loop
`end`	end of if statement checking the block
`%If there is a value for f then this would need to be added`	
`%to the equation below`	
`Bottom=alpha(1,(l1+1))-(((l1/(Last*jay))^(q))*(gamma((-1)*q))*B);`	Calculation of element of algorithm
`Frac=(1/Bottom);`	Calculation of element of algorithm
`if labindex==1`	If condition inspecting the LABNUMBER
`if l1==1`	If condition for step = 1 (i.e. block 1)
`sumI=alpha(2,2)*x(1,1);`	Multiplication of initial condition with final alpha value
`end`	end of if condition
`FinalSumI=((-1)*sumI)-((1/q)*(x(1,1)));`	Calculation of element of algorithm
`x((l1+1),1)=Frac*FinalSumI;`	Final calculation of algorithm at this step utilising previous results
`xx1=labBroadcast(1,x((l1+1),1));`	Broadcast result from LAB1 to others in pool
`if (l1+7)<=(Last*jay)`	If condition for inspecting whether the last step in the block is less than the total number of steps
`xx2=labBroadcast(2);`	Receive broadcast from LAB2
`x((l1+2),1)=xx2;`	Store broadcast from LAB2
`xx3=labBroadcast(3);`	Receive broadcast from LAB3
`x((l1+3),1)=xx3;`	Store broadcast from LAB3
`xx4=labBroadcast(4);`	Receive broadcast from LAB4
`x((l1+4),1)=xx4;`	Store broadcast from LAB4
`xx5=labBroadcast(5);`	Receive broadcast from LAB5
`x((l1+5),1)=xx5;`	Store broadcast from LAB5
`xx6=labBroadcast(6);`	Receive broadcast from LAB6
`x((l1+6),1)=xx6;`	Store broadcast from LAB6
`xx7=labBroadcast(7);`	Receive broadcast from LAB7
`x((l1+7),1)=xx7;`	Store broadcast from LAB7
`xx8=labBroadcast(8);`	Receive broadcast from LAB8
`x((l1+8),1)=xx8;`	Store broadcast from LAB8

Code	Commentary
`elseif (l1+7)==((Last*jay)+1)`	Alternative to if condition. Inspects the last step value in block to see if it is one more than total number of steps
`xx2=labBroadcast(2);`	Receive broadcast from LAB2
`x((l1+2),1)=xx2;`	Store broadcast from LAB2
`xx3=labBroadcast(3);`	Receive broadcast from LAB3
`x((l1+3),1)=xx3;`	Store broadcast from LAB3
`xx4=labBroadcast(4);`	Receive broadcast from LAB4
`x((l1+4),1)=xx4;`	Store broadcast from LAB4
`xx5=labBroadcast(5);`	Receive broadcast from LAB5
`x((l1+5),1)=xx5;`	Store broadcast from LAB5
`xx6=labBroadcast(6);`	Receive broadcast from LAB6
`x((l1+6),1)=xx6;`	Store broadcast from LAB6
`xx7=labBroadcast(7);`	Receive broadcast from LAB7
`x((l1+7),1)=xx7;`	Store broadcast from LAB7
`elseif (l1+7)==((Last*jay)+2)`	Alternative to if statement. Inspects to see if last step in block is two more than the total number of steps
`xx2=labBroadcast(2);`	Receive broadcast from LAB2
`x((l1+2),1)=xx2;`	Store broadcast from LAB2
`xx3=labBroadcast(3);`	Receive broadcast from LAB3
`x((l1+3),1)=xx3;`	Store broadcast from LAB3
`xx4=labBroadcast(4);`	Receive broadcast from LAB4
`x((l1+4),1)=xx4;`	Store broadcast from LAB4
`xx5=labBroadcast(5);`	Receive broadcast from LAB5
`x((l1+5),1)=xx5;`	Store broadcast from LAB5
`xx6=labBroadcast(6);`	Receive broadcast from LAB6
`x((l1+6),1)=xx6;`	Store broadcast from LAB6
`elseif (l1+7)==((Last*jay)+3)`	Alternative to if statement. Inspects to see if last step in block is three more than the total number of steps
`xx2=labBroadcast(2);`	Receive broadcast from LAB2
`x((l1+2),1)=xx2;`	Store broadcast from LAB2
`xx3=labBroadcast(3);`	Receive broadcast from LAB3
`x((l1+3),1)=xx3;`	Store broadcast from LAB3
`xx4=labBroadcast(4);`	Receive broadcast from LAB4
`x((l1+4),1)=xx4;`	Store broadcast from LAB4
`xx5=labBroadcast(5);`	Receive broadcast from LAB5
`x((l1+5),1)=xx5;`	Store broadcast from LAB5
`elseif (l1+7)==((Last*jay)+4)`	Alternative to if statement. Inspects to see if last step in block is four more than the total number of steps
`xx2=labBroadcast(2);`	Receive broadcast from LAB2
`x((l1+2),1)=xx2;`	Store broadcast from LAB2
`xx3=labBroadcast(3);`	Receive broadcast from LAB3
`x((l1+3),1)=xx3;`	Store broadcast from LAB3
`xx4=labBroadcast(4);`	Receive broadcast from LAB4
`x((l1+4),1)=xx4;`	Store broadcast from LAB4

Code	Commentary
elseif (l1+7)==((Last*jay)+5)	Alternative to if statement. Inspects to see if last step in block is five more than the total number of steps
xx2=labBroadcast(2);	Receive broadcast from LAB2
x((l1+2),1)=xx2;	Store broadcast from LAB2
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1+3),1)=xx3;	Store broadcast from LAB3
elseif (l1+7)==((Last*jay)+6)	Alternative to if statement. Inspects to see if last step in block is six more than the total number of steps
xx2=labBroadcast(2);	Receive broadcast from LAB2
x((l1+2),1)=xx2;	Store broadcast from LAB2
end	end of if statement for inspecting the last step in block
else	Alternative if statement to LAB==1
if labindex==2	If statement inspecting if labindex ==2
xx1=labBroadcast(1);	Receive broadcast from LAB1
x((l1),1)=xx1;	Store broadcast from LAB1
elseif labindex==3	Alternative if statement to see if LAB==3
xx1=labBroadcast(1);	Receive broadcast from LAB1
x((l1-1),1)=xx1;	Store broadcast from LAB1
xx2=labBroadcast(2);	Receive broadcast from LAB2
x((l1),1)=xx2;	Store broadcast from LAB2
elseif labindex==4	Alternative if statement to see if LAB==4
xx1=labBroadcast(1);	Receive broadcast from LAB1
x((l1-2),1)=xx1;	Store broadcast from LAB1
xx2=labBroadcast(2);	Receive broadcast from LAB2
x((l1-1),1)=xx2;	Store broadcast from LAB2
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1),1)=xx3;	Store broadcast from LAB3
elseif labindex==5	Alternative if statement to see if LAB==5
xx1=labBroadcast(1);	Receive broadcast from LAB1
x((l1-3),1)=xx1;	Store broadcast from LAB1
xx2=labBroadcast(2);	Receive broadcast from LAB2
x((l1-2),1)=xx2;	Store broadcast from LAB2
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1-1),1)=xx3;	Store broadcast from LAB3
xx4=labBroadcast(4);	Receive broadcast from LAB4
x((l1),1)=xx4;	Store broadcast from LAB4
elseif labindex==6	Alternative if statement to see if LAB==6
xx1=labBroadcast(1);	Receive broadcast from LAB1
x((l1-4),1)=xx1;	Store broadcast from LAB1
xx2=labBroadcast(2);	Receive broadcast from LAB2
x((l1-3),1)=xx2;	Store broadcast from LAB2
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1-2),1)=xx3;	Store broadcast from LAB3
xx4=labBroadcast(4);	Receive broadcast from LAB4
x((l1-1),1)=xx4;	Store broadcast from LAB4
xx5=labBroadcast(5);	Receive broadcast from LAB5

Code	Commentary
x((l1),1)=xx5;	Store broadcast from LAB5
elseif labindex==7	Alternative if statement to see if LAB==7
xx1=labBroadcast(1);	Receive broadcast from LAB1
x((l1-5),1)=xx1;	Store broadcast from LAB1
xx2=labBroadcast(2);	Receive broadcast from LAB2
x((l1-4),1)=xx2;	Store broadcast from LAB2
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1-3),1)=xx3;	Store broadcast from LAB3
xx4=labBroadcast(4);	Receive broadcast from LAB4
x((l1-2),1)=xx4;	Store broadcast from LAB4
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1-1),1)=xx5;	Store broadcast from LAB5
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1),1)=xx6;	Store broadcast from LAB6
elseif labindex==8	Alternative if statement to see if LAB==8
xx1=labBroadcast(1);	Receive broadcast from LAB1
x((l1-6),1)=xx1;	Store broadcast from LAB1
xx2=labBroadcast(2);	Receive broadcast from LAB2
x((l1-5),1)=xx2;	Store broadcast from LAB2
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1-4),1)=xx3;	Store broadcast from LAB3
xx4=labBroadcast(4);	Receive broadcast from LAB4
x((l1-3),1)=xx4;	Store broadcast from LAB4
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1-2),1)=xx5;	Store broadcast from LAB5
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1-1),1)=xx6;	Store broadcast from LAB6
xx7=labBroadcast(7);	Receive broadcast from LAB7
x((l1),1)=xx7;	Store broadcast from LAB7
end	End of if statement for inspecting LABINDEX
%check this bit	
if l1<(numlabs+1)	If statement inspecting if current step is less than 9
for hhh=2:(l1+2)	Loop for adding results from current block to sumI
sumI=sumI+(alpha((l1+3-hhh),(l1+1))*(x((hhh-1),1)));	Cumulative sum of results from other LABs
%sumI=sumI+(alpha((l1-hhh+2),(l1+1))*(x((hhh),1)));	
end	end of for loop for cumulative sum
else	Alternative if statement for current step greater or equal to 9
for hhh=(l1-(labindex-2)):l1	Loop for adding results from current block to sumI
sumI=sumI+(alpha((l1+2-hhh),(l1+1))*(x((hhh),1)));	Cumulative sum of results from other LABs
end	end of for loop for cumulative sum
end	End of if statement inspecting current step

Code	Commentary
FinalSumI=((-1)*sumI)-((1/q)*(x(1,1)));	Calculation of element of algorithm
x((l1+1),1)=Frac*FinalSumI;	Final calculation of algorithm at this step utilising previous results
if labindex==2	If statement inspecting the labindex
if l1==(Last*jay)	If statement inspecting the current step for value = to total number of steps
xx2=labBroadcast(2,x((l1+1),1));	Broadcast result from LAB2 to others in pool
elseif (l1+1)==(Last*jay)	Alternative to if statement when current step + 3 is equal to total number of steps
xx2=labBroadcast(2,x((l1+1),1));	Broadcast result from LAB2 to others in pool
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1+2),1)=xx3;	Store broadcast from LAB3
elseif (l1+2)==(Last*jay)	Alternative to if statement when current step + 2 is equal to total number of steps
xx2=labBroadcast(2,x((l1+1),1));	Broadcast result from LAB2 to others in pool
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1+2),1)=xx3;	Store broadcast from LAB3
xx4=labBroadcast(4);	Receive broadcast from LAB4
x((l1+3),1)=xx4;	Store broadcast from LAB4
elseif (l1+3)==(Last*jay)	Alternative to if statement when current step + 3 is equal to total number of steps
xx2=labBroadcast(2,x((l1+1),1));	Broadcast result from LAB2 to others in pool
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1+2),1)=xx3;	Store broadcast from LAB3
xx4=labBroadcast(4);	Receive broadcast from LAB4
x((l1+3),1)=xx4;	Store broadcast from LAB4
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1+4),1)=xx5;	Store broadcast from LAB5
elseif (l1+4)==(Last*jay)	Alternative to if statement when current step + 4 is equal to total number of steps
xx2=labBroadcast(2,x((l1+1),1));	Broadcast result from LAB2 to others in pool
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1+2),1)=xx3;	Store broadcast from LAB3
xx4=labBroadcast(4);	Receive broadcast from LAB4
x((l1+3),1)=xx4;	Store broadcast from LAB4

Code	Commentary
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1+4),1)=xx5;	Store broadcast from LAB5
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1+5),1)=xx6;	Store broadcast from LAB6
elseif (l1+5)==(Last*jay)	Alternative to if statement when current step + 5 is equal to total number of steps
xx2=labBroadcast(2,x((l1+1),1));	Broadcast result from LAB2 to others in pool
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1+2),1)=xx3;	Store broadcast from LAB3
xx4=labBroadcast(4);	Receive broadcast from LAB4
x((l1+3),1)=xx4;	Store broadcast from LAB4
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1+4),1)=xx5;	Store broadcast from LAB5
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1+5),1)=xx6;	Store broadcast from LAB6
xx7=labBroadcast(7);	Receive broadcast from LAB7
x((l1+6),1)=xx7;	Store broadcast from LAB7
else	Alterative to if statement when all other scenarios are invalid i.e. when all LABS are active in block
xx2=labBroadcast(2,x((l1+1),1));	Broadcast result from LAB2 to others in pool
xx3=labBroadcast(3);	Receive broadcast from LAB3
x((l1+2),1)=xx3;	Store broadcast from LAB3
xx4=labBroadcast(4);	Receive broadcast from LAB4
x((l1+3),1)=xx4;	Store broadcast from LAB4
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1+4),1)=xx5;	Store broadcast from LAB5
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1+5),1)=xx6;	Store broadcast from LAB6
xx7=labBroadcast(7);	Receive broadcast from LAB7
x((l1+6),1)=xx7;	Store broadcast from LAB7
xx8=labBroadcast(8);	Receive broadcast from LAB8
x((l1+7),1)=xx8;	Store broadcast from LAB8
end	End of if statement inspecting current step
elseif labindex==3	Alternative to if statement when LAB==3

Code	Commentary
`if l1==(Last*jay)`	If statement inspecting the current step for value = to total number of steps
`xx3=labBroadcast(3,x((l1+1),1));`	Broadcast result from LAB3 to others in pool
`elseif (l1+1)==(Last*jay)`	Alternative to if statement when current step + 1 is equal to total number of steps
`xx3=labBroadcast(3,x((l1+1),1));`	Broadcast result from LAB3 to others in pool
`xx4=labBroadcast(4);`	Receive broadcast from LAB4
`x((l1+2),1)=xx4;`	Store broadcast from LAB4
`elseif (l1+2)==(Last*jay)`	Alternative to if statement when current step + 2 is equal to total number of steps
`xx3=labBroadcast(3,x((l1+1),1));`	Broadcast result from LAB3 to others in pool
`xx4=labBroadcast(4);`	Receive broadcast from LAB4
`x((l1+2),1)=xx4;`	Store broadcast from LAB4
`xx5=labBroadcast(5);`	Receive broadcast from LAB5
`x((l1+3),1)=xx5;`	Store broadcast from LAB5
`elseif (l1+3)==(Last*jay)`	Alternative to if statement when current step + 3 is equal to total number of steps
`xx3=labBroadcast(3,x((l1+1),1));`	Broadcast result from LAB3 to others in pool
`xx4=labBroadcast(4);`	Receive broadcast from LAB4
`x((l1+2),1)=xx4;`	Store broadcast from LAB4
`xx5=labBroadcast(5);`	Receive broadcast from LAB5
`x((l1+3),1)=xx5;`	Store broadcast from LAB5
`xx6=labBroadcast(6);`	Receive broadcast from LAB6
`x((l1+4),1)=xx6;`	Store broadcast from LAB6
`elseif (l1+4)==(Last*jay)`	Alternative to if statement when current step + 4 is equal to total number of steps
`xx3=labBroadcast(3,x((l1+1),1));`	Broadcast result from LAB3 to others in pool
`xx4=labBroadcast(4);`	Receive broadcast from LAB4
`x((l1+2),1)=xx4;`	Store broadcast from LAB4
`xx5=labBroadcast(5);`	Receive broadcast from LAB5
`x((l1+3),1)=xx5;`	Store broadcast from LAB5
`xx6=labBroadcast(6);`	Receive broadcast from LAB6
`x((l1+4),1)=xx6;`	Store broadcast from LAB6
`xx7=labBroadcast(7);`	Receive broadcast from LAB7
`x((l1+5),1)=xx7;`	Store broadcast from LAB7

Code	Commentary
else	Alternative to if statement when all other scenarios are invalid i.e. when all LABs are active in block
xx3=labBroadcast(3,x((l1+1),1));	Broadcast result from LAB3 to others in pool
xx4=labBroadcast(4);	Receive broadcast from LAB4
x((l1+2),1)=xx4;	Store broadcast from LAB4
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1+3),1)=xx5;	Store broadcast from LAB5
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1+4),1)=xx6;	Store broadcast from LAB6
xx7=labBroadcast(7);	Receive broadcast from LAB7
x((l1+5),1)=xx7;	Store broadcast from LAB7
xx8=labBroadcast(8);	Receive broadcast from LAB8
x((l1+6),1)=xx8;	Store broadcast from LAB8
end	End of if statement inspecting current step
elseif labindex==4	Alternative to if statement applicable to LAB ==4
if l1==(Last*jay)	If statement inspecting the current step for value = to total number of steps
xx4=labBroadcast(4,x((l1+1),1));	Broadcast result from LAB4 to others in pool
elseif (l1+1)==(Last*jay)	Alternative to if statement when current step + 1 is equal to total number of steps
xx4=labBroadcast(4,x((l1+1),1));	Broadcast result from LAB4 to others in pool
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1+2),1)=xx5;	Store broadcast from LAB5
elseif (l1+2)==(Last*jay)	Alternative to if statement when current step + 2 is equal to total number of steps
xx4=labBroadcast(4,x((l1+1),1));	Broadcast result from LAB4 to others in pool
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1+2),1)=xx5;	Store broadcast from LAB5
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1+3),1)=xx6;	Store broadcast from LAB6
elseif (l1+3)==(Last*jay)	Alternative to if statement when current step + 3 is equal to total number of steps
xx4=labBroadcast(4,x((l1+1),1));	Broadcast result from LAB4 to others in pool
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1+2),1)=xx5;	Store broadcast from LAB5

Code	Commentary
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1+3),1)=xx6;	Store broadcast from LAB6
xx7=labBroadcast(7);	Receive broadcast from LAB7
x((l1+4),1)=xx7;	Store broadcast from LAB7
else	Alternative to if statement when all other scenarios are invalid i.e. when all LABs are active in block
xx4=labBroadcast(4,x((l1+1),1));	Broadcast result from LAB4 to others in pool
xx5=labBroadcast(5);	Receive broadcast from LAB5
x((l1+2),1)=xx5;	Store broadcast from LAB5
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1+3),1)=xx6;	Store broadcast from LAB6
xx7=labBroadcast(7);	Receive broadcast from LAB7
x((l1+4),1)=xx7;	Store broadcast from LAB7
xx8=labBroadcast(8);	Receive broadcast from LAB8
x((l1+5),1)=xx8;	Store broadcast from LAB8
end	End of if statement inspecting current step
elseif labindex==5	Alternative if statement applicable to LAB==5
if l1==(Last*jay)	If statement inspecting the current step for value = to total number of steps
xx5=labBroadcast(5,x((l1+1),1));	Broadcast result from LAB5 to others in pool
elseif (l1+1)==(Last*jay)	Alternative to if statement when current step + 1 is equal to total number of steps
xx5=labBroadcast(5,x((l1+1),1));	Broadcast result from LAB5 to others in pool
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1+2),1)=xx6;	Store broadcast from LAB6
elseif (l1+2)==(Last*jay)	Alternative to if statement when current step + 2 is equal to total number of steps
xx5=labBroadcast(5,x((l1+1),1));	Broadcast result from LAB5 to others in pool
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1+2),1)=xx6;	Store broadcast from LAB6
xx7=labBroadcast(7);	Receive broadcast from LAB7
x((l1+3),1)=xx7;	Store broadcast from LAB7
else	Alternative to if statement when current step + 3 is equal to total number of steps
xx5=labBroadcast(5,x((l1+1),1));	Broadcast result from LAB5 to others in pool

Code	Commentary
xx6=labBroadcast(6);	Receive broadcast from LAB6
x((l1+2),1)=xx6;	Store broadcast from LAB6
xx7=labBroadcast(7);	Receive broadcast from LAB7
x((l1+3),1)=xx7;	Store broadcast from LAB7
xx8=labBroadcast(8);	Receive broadcast from LAB8
x((l1+4),1)=xx8;	Store broadcast from LAB8
end	End of if statement inspecting current step
elseif labindex==6	Alternative to if statement applicable to LAB ==6
if l1==(Last*jay)	If statement inspecting the current step for value = to total number of steps
xx6=labBroadcast(6,x((l1+1),1));	Broadcast result from LAB6 to others in pool
elseif (l1+1)==(Last*jay)	Alternative to if statement when current step + 1 is equal to total number of steps
xx6=labBroadcast(6,x((l1+1),1));	Broadcast result from LAB6 to others in pool
xx7=labBroadcast(7);	Receive broadcast from LAB7
x((l1+2),1)=xx7;	Store broadcast from LAB7
else	Alternative to if statement when current step + 2 is equal to total number of steps
xx6=labBroadcast(6,x((l1+1),1));	Broadcast result from LAB6 to others in pool
xx7=labBroadcast(7);	Receive broadcast from LAB7
x((l1+2),1)=xx7;	Store broadcast from LAB7
xx8=labBroadcast(8);	Receive broadcast from LAB8
x((l1+3),1)=xx8;	Store broadcast from LAB8
end	End of if statement inspecting current step
elseif labindex==7	Alternative to if statement applicable to LAB==7
if l1==(Last*jay)	If statement inspecting the current step for value = to total number of steps
xx7=labBroadcast(7,x((l1+1),1));	Broadcast result from LAB7 to others in pool
else	Alternative to if statement when current step + 1 is equal to total number of steps
xx7=labBroadcast(7,x((l1+1),1));	Broadcast result from LAB7 to others in pool
xx8=labBroadcast(8);	Receive broadcast from LAB8
x((l1+2),1)=xx8;	Store broadcast from LAB8
end	End of if statement inspecting current step
else	Alternative to if statement applicable to LAB8 only
xx8=labBroadcast(8,x((l1+1),1));	Broadcast result from LAB8 to others in pool
end	End of if statement inspecting the LABINDEX

Code	Commentary
`end`	End of if statement beginning Labindex==1
`elseif l1>(Last*jay)`	Alternative if statement when current step is more than the total number of steps required to complete algorithm
`disp('The end of the calculation has been achieved')`	Display text
`end`	End of if statement beginning if l1<=(Last*jay)
`end`	End loop for operating in blocks
`% if labindex==1`	
`% disp('The value of x is...')`	
`% disp(x)`	
`% else`	
`disp('Calculations are complete')`	Display text
`% end`	
`end`	End of SPMD command
`matlabpool close`	Close the MatLab Pool
`toc`	Stop the clock and display time
`end`	End of for loop to operate the program 13 times

C. FRACTIONAL ADAMS METHOD (FAM) - ANNOTATED PROGRAM

Following work in Chapter 9 an annotated version of the program 'ABMegDv5_2' is provided below. The program operates on a four-worker pool and utilises a 'block' approach (see Section 4.5). 'ABMegDv5_2' contains merged coefficient calculations undertaken by each worker. Additionally a series of 'if statements' are provided so that 'idle workers' can accept broadcasts from 'active workers' in the pool. This 'idle'/'active' worker scenario can occur in the final block of the program where the number of steps taken is not divisible by the number of workers (in this case, four).

Code	Commentary
`% Restructured ABM program following Diethelm paper.`	-
`T=input('Enter the last variable');`	User input for termination point
`%This is what it should be T=5;`	-
`n=input('Enter the number of steps within one unit');`	User input for total number of steps
`%This is what it should be n=2*(10^5);`	-
`%alpha=input('Enter the value of the fractional derivative');`	-
`alpha=1.3;`	Value of fractional derivative
`%initial=input('Enter the initial value array');`	-
`initial=[1,0];`	Initial value for FDE
`%Aggregated coefficient matrix, a=abc[:,1], b=abc[:,2], c=abc[:,3]`	-
`for cloop=1:13`	Loop for repeating calculation 13 times for averaging purposes
`disp('Loop')`	Display text
`disp(cloop)`	Display the loop number
`tic`	Begin timing
`h=1/n;`	Calculate the step size
`matlabpool 4`	Open the matlabpool
`abc=zeros((T*n),2);`	Create array for coefficients at intermediate steps
`y=zeros(((T*n)+1),2);`	Create array for solutions at intermediate steps.
`y(1,2)=initial(1);`	Store the initial value within y array
`spmd`	Single program multiple data command
`for endval=1:ceil((T*n)/numlabs)`	Loop which breaks the steps into blocks for the 4 LABs to work on
`l1=(numlabs*(endval-1))+labindex;`	Calculates the step which a LAB should work on
`j=l1;`	Assigns the value of the step to new variable j
`if j<=(T*n)`	Check if the step j is less than the total steps needed to complete the algorithm
`for i=1:j`	Outer loop for calculating coefficients based upon current step LAD is working on
`if i==1`	Inner loop for calculating coefficients
`%This is a coefficient`	-
`abc(i,1)=(((j - 1)^(1 + alpha)) - ((j - alpha - 1)*(j^(alpha))))/(gamma(2 + alpha));`	Calculation of coefficent a_{oj} in current step
`else`	Alternative calculation for other 'a' coefficient values
`abc(i,1)=(((j - (i-1) + 1)^(1 + alpha)) + ((j - (i-1) - 1)^(1 + alpha)) - (2*((j-(i-1))^(1 + alpha))))/(gamma(2 + alpha));`	Calculation of coefficent a_{ij} in current step
`end`	End of 'if'statement beginning if i==1

Code	Commentary
%This is b coefficient	-
abc(i,2)=(((j-(i-1))^(alpha))- ((j-1-(i-1))^(alpha)))/(gamma(1 + alpha));	Calculation of coefficient b_{ij} in current step
end	End of for loop beginning for i=1:j
sumII=0;	Assign 0 to sumII
hp=0;	Assign 0 to hp
sumacy=0;	Assign 0 to sumacy
for k=1:(ceil(alpha))	Loop running from 1 to rounded up value of fractional derivative
sumII = sumII + ((((h*j)^(k-1))/(factorial(k-1)))*(initial(k)));	Cumulative sum. Initial value aspects of the predictor and corrector formula.
end	end of loop (k)
if endval==1	If' statement applicable to block 1 only
hp=(abc(1,2)*((-1)*(y(1,2))));	Assignment of hp using initial condition data
sumacy=(abc(1,1)*((-1)*(y(1,2))));	Assignment of sumacy using initial condition data
else	Alternative when block is not equal to 1
for h1=1:(((endval-1)*4)+1)	Loop for cumulative sum of previous steps for predictor formula.
%The last part of the equation will change for	-
%different FDE.	-
hp=hp+(abc(h1,2)*((-1)*(y(h1,2))));	Cumulative sum of previous steps in predictor formula.
sumacy=sumacy+(abc(h1,1)*((-1)*(y(h1,2))));	Cumulative sum of previous steps multiplied by a_u weight
end	end of loop (h1)
end	End of 'if' statement inspecting block number
if labindex==1	Decide if LAB = 1
y((j+1),1)=sumII+((h^(alpha))*hp);	Calculate the predicted value at step
hypart = sumacy + ((1/(gamma(2+alpha)))*((-1)*y((j+1),1)));	Add previous step calculation to the predicted value at current step
y((j+1),2)=sumII+((h^(alpha))*hypart);	Final calculation for solution at current step
yy1=labBroadcast(1,y((j+1),2));	Broadcast result to other LABs
if (j+3)<=(T*n)	Operate if all other LABs are active
yy2=labBroadcast(2);	Receive broadcast from LAB 2
y((j+2),2)=yy2;	store result from LAB 2

Code	Commentary
`yy3=labBroadcast(3);`	receive broadcast from LAB 3
`y((j+3),2)=yy3;`	store result from LAB 3
`yy4=labBroadcast(4);`	receive broadcast from LAB 4
`y((j+4),2)=yy4;`	store result from LAB 4
`elseif (j+3)==((T*n)+1)`	Decide if all LABs but LAB 4 are active
`yy2=labBroadcast(2);`	Receive broadcast from LAB 2
`y((j+2),2)=yy2;`	store result from LAB 2
`yy3=labBroadcast(3);`	receive broadcast from LAB 3
`y((j+3),2)=yy3;`	store result from LAB 3
`elseif (j+3)==((T*n)+2)`	If only LAB 2 is active (outside of LAB 1!)
`yy2=labBroadcast(2);`	Receive broadcast from LAB 2
`y((j+2),2)=yy2;`	store result from LAB 2
`end`	end of if statement beginning 'j+3<=(T*n)'
`else`	Alternative to 'if labindex==1'
`if labindex==2`	Decide if LAB = 2
`yy1=labBroadcast(1);`	Receive broadcast from LAB 1
`y(j,2)=yy1;`	Store broadcast from LAB 1
`elseif labindex==3`	Decide if LAB = 3
`yy1=labBroadcast(1);`	Receive broadcast from LAB 1
`y((j-1),2)=yy1;`	Store broadcast from LAB 1
`yy2=labBroadcast(2);`	Receive broadacst from LAB 2
`y(j,2)=yy2;`	Store broadcast from LAB 2
`else`	Alternative option (applicable to LAB 4)
`yy1=labBroadcast(1);`	Receive broadcast from LAB 1
`y((j-2),2)=yy1;`	Store broadcast from LAB 1
`yy2=labBroadcast(2);`	Receive broadacst from LAB 2
`y((j-1),2)=yy2;`	Store broadcast from LAB 2
`yy3=labBroadcast(3);`	Receive broadcast from LAB 3

Code	Commentary
`y(j,2)=yy3;`	Store broadcast from LAB 3
`end`	end of if statement beginning 'labindex = 2'
`for h2=(((endval-1)*4)+2):j`	Loop for previous y values calculated by other LABs, these are added to the cumulative sums hp and sumacy
`hp=hp+(abc(h2,2)*((-1)*(y(h2,2))));`	Add other LABs corrector values to cumulative sum hp
`sumacy=sumacy+(abc(h2,1)*((-1)*(y(h2,2))));`	Add other LABs corrector values to cumulative sum sumacy
`end`	end of loop beginning 'h1=(j-labindex-2):j'
`y((j+1),1)=sumII+((h^(alpha))*hp);`	predict the value of y at current step
`hypart = sumacy + ((1/(gamma(2+alpha)))*((-1)*y((j+1),1)));`	Add previous step calculations (sumacy) to the predicted value at current step
`%The last part of the equation will change for different FDE.`	-
`y((j+1),2)=sumII+((h^(alpha))*hypart);`	Corrector value of y at current step
`if labindex==2`	Decide if labindex =2
`if j==(T*n)`	Decide if current step is the last step requiring calculation
`yy2=labBroadcast(2,y((j+1),2));`	Broadcast result to other LABs
`elseif (j+1)==(T*n)`	Decide if the step in front of current is equal to the last step requiring calculation
`yy2=labBroadcast(2,y((j+1),2));`	Broadcast result to other LABs
`yy3=labBroadcast(3);`	Receive broadcast from LAB 3
`y((j+2),2)=yy3;`	Store broadcast from LAB 3
`else`	Alternative when all other options are not valid
`yy2=labBroadcast(2,y((j+1),2));`	Broadcast result to other LABs
`yy3=labBroadcast(3);`	Receive broadcast from LAB 3
`y((j+2),2)=yy3;`	Store broadcast from LAB 3
`yy4=labBroadcast(4);`	Receive broadcast from LAB 4
`y((j+3),2)=yy4;`	Store broadcast from LAB 4
`end`	end of if statement beginning 'if j==(T*n)'
`elseif labindex==3`	Decide if LAB = 3
`if j==(T*n)`	Decide if current step is equal to the total steps requiring calculation

Code	Commentary
`yy3=labBroadcast(3,y((j+1),2));`	Broadcast result to other LABs
`else`	Alternative when current step is not equal to the total number of steps requiring calculation
`yy3=labBroadcast(3,y((j+1),2));`	Broadcast result to other LABs
`yy4abc=labBroadcast(4);`	Receive broadcast from LAB 4
`yabc((j+2),:)=yy4abc;`	Store broadcast from LAB 4
`end`	end of if statement beginning 'if j<(T*n)'
`else`	Alternative to other options i.e. LAB = 4
`yy4=labBroadcast(4,y((j+1),2));`	Broadcast result to other LABs
`end`	end of if statement beginning 'if labindex==2'
`end`	end of if statement beginning 'if labindex==1'
`elseif j>(T*n)`	Alternative to if current step j <=(T*n)
`disp('The end of the calculation has been achieved')`	Display text
`end`	End of if statement beginning 'if j<=(T*n)'
`end`	End of for loop for block programming
`% if labindex==1`	-
`% if cloop==1`	-
`% disp('The value of yabc is...')`	-
`% disp(yabc)`	-
`% end`	-
`% disp('Calculations are complete')`	-
`% else`	-
`% disp('Calculations are complete')`	-
`% end`	-
`end`	End of SPMD
`matlabpool close`	Close matlab pool
`toc`	Display time for execution
`end`	End of for loop beginning 'cloop=1:13'

D. LUBICH'S FRACTIONAL MULTISTEP METHOD (FMM) - ANNOTATED PROGRAM

Following the simplified code provided in Chapter 10 an annotated version of the program 'LubichParallel8Labsv4INLOOPAttemptToControlThreads' is provided. This program is structured as the simplified code in Section 10.3, where only one thread is in operation during the initial phase (calculation of weights etc) prior to a parallel phase utilising eight workers or LABs within the architecture.

CODE	COMMENTARY
%This is the beginning of the Lubich Method	-
% T=input('Enter the termination point, T');	-
T=1;	Termination Point
h=input('Enter the step, h ');	User input of step size
alpha=0.25;	Value of fractional derivative
%Trapezium Rule is the underlying numerical method, s will always be 4 with	-
%this method and alpha = 0.25	-
s=4;	Limit to starting phase equation (card *A* - 1)
for cloop=1:13	Start of loop to run program 13 times
LASTN = maxNumCompThreads(1);	
disp('Loop')	Displays text to explain following value
disp(cloop)	Displays which loop program currently working on
tic;	Starts the timer
N=T/h;	Total number of steps in algorithm
%if the Matlabpool is different than four, the value in x dimensions would	-
%be different	-
x=zeros(2,(N+1));	Creation of x matrix for storing calculated x values (first row) and functional results (second row).
x(1,1)=0;	Setting the initial condition into the matrix
% Coefficients for the predictor	-
startcoef=zeros(1,4);	Creation of array for starting coefficients used in prediction of x (this is *Beta* variable).
for i=1:N	Loop used in calcuating the starting coefficients
startcoef(1,i)=((i^alpha)-((i-1)^alpha))/(gamma(alpha+1));	Calculating the starting coefficient at point I (this is *Beta* variable)
end	End of loop for starting coefficients
%%%%%%%%%%Calculation of characterist polynomial	-
Aarray=zeros(1,N+1);	Array used in calculating the characteristic polynomial and therefore convolution coefficients
Barray=zeros(1,N+1);	Array used in calculating the characteristic polynomial and therefore convolution coefficients
ABMatrix=zeros(N+1);	Matrix used for storing results of multiplying Aarray and Barray together
Apart=1;	Initial value of Apart
Bpart=1;	Initial value of Bpart
Ocount=0;	Zero the value of Ocount
Ecount=0;	Zero the value of Ecount
for j=1:(N+1)	Loop used in calculating the convolution weights

`if j==1`	First loop only
`Aarray(1)=1;`	Setting the initial value in Aarray
`Barray(1)=1;`	Setting the initial value in Barray
`elseif (j~=1 && rem(j,2)==1)`	Does not equal to one and is a loop of odd number
`Ocount=Ocount+1;`	Add 1 to the existing value of Ocount
`Apart=Apart*((alpha-(Ocount-1))/Ocount);`	Use of existing value of Apart (calculated in previous applicable step or assigned) and multiplied by next element of binomial expansion
`Aarray(j)=Apart;`	Assign Apart to the current Aarray element
`Ecount=Ecount+1;`	Add 1 to the existing value of Ecount
`Bpart=Bpart*((alpha+(Ecount-1))/Ecount);`	Use of existing value of Bpart (calculated in previous step or assigned) and multiplied by next element of binomial expansion
`Barray(j)=Bpart;`	Assign Bpart to the current Barray element
`elseif (j~=1 && rem(j,2)==0)`	Does not equal to one and is a loop of even number
`Ecount=Ecount+1;`	Add 1 to the existing value of Ecount
`Bpart=Bpart*((alpha+(Ecount-1))/Ecount);`	Use of existing value of Bpart (calculated in previous step or assigned) and multiplied by next element of binomial expansion
`Barray(j)=Bpart;`	Assign Bpart to the current Barray element
`Aarray(j)=0;`	Current value of Aarray is zero
`end`	End of **if command** which begins j==1
`end`	End of loop which begins j=1:(N+1)
`for jv=1:(N+1)`	Loop use to multiply Aarray and Barray together
`for kv=1:((N+1)-(jv-1))`	In current loop, multiply all elements of Aarray together to current Barray element
`ABMatrix(jv,(jv+(kv-1)))=Barray(jv)*Aarray(kv);`	Multiplying Barray and Aarray components and storing result in ABMatrix
`end`	End of loop which begins kv=1:((N+1)-(jv-1))
`end`	End of loop which begins jv=1:(N+1)
`ABMatrix=(1/(2^alpha))*ABMatrix;`	Final calculation of the ABMatrix
`convweightsT=sum(ABMatrix);`	Sums the columns of the matrix ABMatrix
`convweights=convweightsT.';`	Transposes the matrix for use in the algorithm.
`LASTN = maxNumCompThreads('automatic');`	
`%The starting phase and the Matlabpool (plus initial condition) are same in this example`	-
`%  parpool (8)`	Alternative way to open MatLab Pool, used in newer versions of MatLab Parallel Computing Toolbox
`matlabpool 8`	Beginning communication with 8 Labs in the MatLab Pool
`spmd`	Single Program Multiple Data command
`for block=1:(ceil(N/8))`	Loop which splits the steps into blocks for the 8 Labs to work on

`n=labindex+(8*(block-1));`	Assigns the data for the lab to work on, specific for each lab within the block
`if n<=N`	Only continues with this part of program when n value is not larger than the total number of steps
`%Calculation of the values for the starting weights`	-
`%If wanted a different alpha this part would need to be altered`	-
`Tots=zeros(1,4);`	Set up of array for calculating convolution component for starting weights calculations (in starting phase)
`Tots(1,1)=0;`	NOT NEEDED
`Tots(1,2)=0;`	NOT NEEDED
`Tots(1,3)=0;`	NOT NEEDED
`Tots(1,4)=0;`	NOT NEEDED
`Totconvweights=0;`	Assigning zero value to Totconvweights
`for k=1:(n+1)`	Loop for calculating the starting weights in starting phase, loop ends at current n value plus 1
`Tots(1,1)=Tots(1,1)+((n-(k-1))^(alpha))*(convweights(k,1));`	Cumulative total for Tot(1,1) + component from convolution weights
`Tots(1,2)=Tots(1,2)+((n-(k-1))^(2*alpha))*(convweights(k,1));`	Cumulative total for Tot(1,2) + component from convolution weights
`Tots(1,3)=Tots(1,3)+((n-(k-1))^(3*alpha))*(convweights(k,1));`	Cumulative total for Tot(1,3) + component from convolution weights
`Tots(1,4)=Tots(1,4)+((n-(k-1))^(4*alpha))*(convweights(k,1));`	Cumulative total for Tot(1,4) + component from convolution weights
`Totconvweights=Totconvweights+convweights(k,1);`	Cumulative total for Totconvweights
`end`	End of loop which begins k=1:(n+1)
`A=((1/(gamma(alpha)))*((n^alpha)/alpha))-Totconvweights;`	Answer part of starting weight calculation for starting phase (first simultaneous equation)
`E=((1/(gamma(alpha)))*((n^(1+alpha))/(alpha*(1+alpha))))-(Tots(1,4));`	Answer part of starting weight calculation for starting phase (final simultaneous equation)
`Z=(((-1)*(Tots(1,2)))+Tots(1,1))/((2^(2*alpha))-(2^alpha));`	Component for calculating the starting weight for starting phase
`X=((3^(2*alpha))-(3^alpha))/((2^(2*alpha))-(2^alpha));`	Component for calculating the starting weight for starting phase
`Y=((4^(2*alpha))-(4^alpha))/((2^(2*alpha))-(2^alpha));`	Component for calculating the starting weight for starting phase
`F=((4^(3*alpha))-(4^alpha))/((2^(3*alpha))-(2^alpha));`	Component for calculating the starting weight for starting phase
`EEE=((3^(3*alpha))-(3^alpha))/((2^(3*alpha))-(2^alpha));`	Component for calculating the starting weight for starting phase
`DD=(F-Y)/(EEE-X);`	Component for calculating the starting weight for starting phase

`G=(((-1)*(Tots(1,3)))+Tots(1,1))/((2^(3*alpha))-(2^alpha));`	Component for calculating the starting weight for starting phase
`EE=(G-Z)/(EEE-X);`	Component for calculating the starting weight for starting phase
`KK=(4-(4^alpha))/(2-(2^alpha));`	Component for calculating the starting weight for starting phase
`J=(3-(3^alpha))/(2-(2^alpha));`	Component for calculating the starting weight for starting phase
`II=(KK-Y)/(J-X);`	Component for calculating the starting weight for starting phase
`L=(E+Tots(1,1))/(2-(2^alpha));`	Component for calculating the starting weight for starting phase
`JJ=(L-Z)/(J-X);`	Component for calculating the starting weight for starting phase
`%Starting weight calculations`	-
`startweight=zeros(1,5);`	Creation of starting weight (for starting phase) matrix
`startweight(1,5)=(JJ-EE)/(II-DD);`	Calculation of fifth starting weight
`startweight(1,4)=EE-(DD*startweight(1,5));`	Calculation of fourth starting weight
`startweight(1,3)=Z-(Y*startweight(1,5))-(X*startweight(1,4));`	Calculation of third starting weight
`startweight(1,2)=((-1)*(Tots(1,1)))-((4^alpha)*startweight(1,5))-((3^alpha)*startweight(1,4))-((2^alpha)*startweight(1,3));`	Calculation of second starting weight
`startweight(1,1)=A-startweight(1,2)-startweight(1,3)-startweight(1,4)-startweight(1,5);`	Calculation of first starting weight
`if (block==1 && labindex<5)`	**If command** for pre-phase, only applies to first block and LABS 1-4
`%Starting pre-phase`	-
`if labindex==1`	**If command** for LAB 1
`%This is the function trying to solve. The value of h has`	-
`%been subbed for zero.`	-
`val1=(40320/(gamma(9-alpha)))*(0^(8-alpha));`	First part of functional value
`val2=3*((gamma(5+(alpha/2)))/(gamma(5-(alpha/2))))*(0^(4-(alpha/2)));`	Second part of functional value
`val3=(9/4)*(gamma(alpha+1));`	Third part of function value
`val4=(((3/2)*(0^(alpha/2)))-(0^4))^3;`	Fourth part of functional value
`x(2,n)=val1-val2+val3+val4-(x(1,1)^(3/2));`	Final calculation of functional value of x at initial condition
`%predictor method follows.`	-
`%No value for the initial part of the predictor, this`	-

%would change with different initial value x(1,1)	-
x(1,(n+1))=(h^alpha)*startcoef(1,n)*x(2,n);	Predictor for x(1,2) using starting coefficient and functional value of previous step
labSend(x(1,(n+1)),2,11);	Send result of x(1,2) to LAB 2
labSend(x(2,n),2,12);	Send functional value x(2,1) to LAB 2
labSend(x(1,(n+1)),3,11);	Send result of x(1,2) to LAB 3
labSend(x(2,n),3,12);	Send functional value x(2,1) to LAB 3
labSend(x(1,(n+1)),4,11);	Send result of x(1,2) to LAB 4
labSend(x(2,n),4,12);	Send functional value x(2,1) to LAB 4
start4=labBroadcast(4);	Receive broadcast from LAB 4
x=start4;	Store result from 4 as new value for x matrix
elseif labindex==2	**If command** for LAB 2
x(1,n)=labReceive(1,11);	Receive data from LAB 1 (x value)
x(2,(n-1))=labReceive(1,12);	Receive data from LAB 1 (function)
xpre=0;	Set xpre (used in predictor calculation) to zero
%Equation trying to solve.	-
val1=(40320/(gamma(9-alpha)))*(h^(8-alpha));	First part of functional value
val2=3*((gamma(5+(alpha/2)))/(gamma(5-(alpha/2))))*(h^(4-(alpha/2)));	Second part of functional value
val3=(9/4)*(gamma(alpha+1));	Third part of function value
val4=(((3/2)*(h^(alpha/2)))-(h^4))^3;	Fourth part of functional value
x(2,n)=val1-val2+val3+val4-((x(1,n))^(3/2));	Final calculation of functional value of x(1,n) step
for ii=1:labindex	**For loop** - cumulative total for predictor, using starting coefficients for predictor multiplied by functional value
%Predictor follows	-
%No value for the initial part of the predictor, this	-
%would change with different initial value x(1,1)	-
xpre=xpre+(startcoef(1,(n+1-ii))*x(2,ii));	Cumulative total for predictive element
end	**End of loop** which began ii=1:labindex
x(1,(n+1))=(h^alpha)*xpre;	Predicted value for x(1,(n+1))
%x(1,(n+1))=(h^alpha)*xpre;	-
labSend(x(1,(n+1)),3,21);	Send result of x(1,(n+1)) to LAB 3
labSend(x(2,n),3,22);	Send functional value x(2,n) to LAB 3
labSend(x(1,(n+1)),4,21);	Send result of x(1,(n+1)) to LAB 4
labSend(x(2,n),4,22);	Send functional value x(2,n) to LAB 4
start4=labBroadcast(4);	Receive broadcast from LAB 4

x=start4;	Store result from 4 as new value for x matrix
elseif labindex==3	**If command** for LAB 3
x(1,(n-1))=labReceive(1,11);	Receive data from LAB 1 (x value)
x(2,(n-2))=labReceive(1,12);	Receive data from LAB 1 (function)
x(1,n)=labReceive(2,21);	Receive data from LAB 2 (x value)
x(2,(n-1))=labReceive(2,22);	Receive data from LAB 2 (function)
xpre=0;	Set xpre (used in predictor calculation) to zero
%Equation trying to solve.	-
val1=(40320/(gamma(9-alpha)))*(((n-1)*h)^(8-alpha));	First part of functional value
val2=3*((gamma(5+(alpha/2)))/(gamma(5-(alpha/2))))*(((n-1)*h)^(4-(alpha/2)));	Second part of functional value
val3=(9/4)*(gamma(alpha+1));	Third part of function value
val4=(((3/2)*(((n-1)*h)^(alpha/2)))-(((n-1)*h)^4))^3;	Fourth part of functional value
x(2,n)=val1-val2+val3+val4-((x(1,n))^(3/2));	Final calculation of functional value of x(1,n) step
for i3=1:labindex	**For loop** - cumulative total for predictor, using starting coefficients for predictor multiplied by functional value
%Predictor follows	-
%No value for the initial part of the predictor, this	-
%would change with different initial value x(1,1)	-
xpre=xpre+(startcoef(1,(n+1-i3))*x(2,i3));	Cumulative total for predictive element
end	**End of loop** which began i3=1:labindex
x(1,(n+1))=(h^alpha)*xpre;	Predicted value for x(1,(n+1))
labSend(x(1,(n+1)),4,31);	Send result of x(1,(n+1)) to LAB 4
labSend(x(2,n),4,32);	Send functional value x(2,n) to LAB 4
start4=labBroadcast(4);	Receive broadcast from LAB 4
x=start4;	Store result from 4 as new value for x matrix
elseif labindex==4	**If command** for LAB 4
x(1,(n-2))=labReceive(1,11);	Receive data from LAB 1 (x value)
x(2,(n-3))=labReceive(1,12);	Receive data from LAB 1 (function)
x(1,(n-1))=labReceive(2,21);	Receive data from LAB 2 (x value)
x(2,(n-2))=labReceive(2,22);	Receive data from LAB 2 (function)
x(1,n)=labReceive(3,31);	Receive data from LAB 3 (x value)

`x(2,(n-1))=labReceive(3,32);`	Receive data from LAB 3 (function)
`xpre=0;`	Set xpre (used in predictor calculation) to zero
`%Equation trying to solve.`	-
`val1=(40320/(gamma(9-alpha)))*(((n-1)*h)^(8-alpha));`	First part of functional value
`val2=3*((gamma(5+(alpha/2)))/(gamma(5-(alpha/2))))*(((n-1)*h)^(4-(alpha/2)));`	Second part of functional value
`val3=(9/4)*(gamma(alpha+1));`	Third part of function value
`val4=(((3/2)*(((n-1)*h)^(alpha/2)))-(((n-1)*h)^4))^3;`	Fourth part of functional value
`x(2,n)=val1-val2+val3+val4-((x(1,n))^(3/2));`	Final calculation of functional value of x(1,n) step
`for i4=1:labindex`	**For loop** - cumulative total for predictor, using starting coefficients for predictor multiplied by functional value
`%Predictor follows`	-
`%No value for the initial part of the predictor, this`	-
`%would change with different initial value x(1,1)`	-
`xpre=xpre+(startcoef(1,(n+1-i4))*x(2,i4));`	Cumulative total for predictive element
`end`	End of loop which began ii=1:labindex
`x(1,(n+1))=(h^alpha)*xpre;`	Predicted value for x(1,(n+1))
`%%%%%%%%%%%Calc x(2,5)`	-
`val1=(40320/(gamma(9-alpha)))*(((block*4)*h)^(8-alpha));`	First part of functional value
`val2=3*((gamma(5+(alpha/2)))/(gamma(5-(alpha/2))))*(((block*4)*h)^(4-(alpha/2)));`	Second part of functional value
`val3=(9/4)*(gamma(alpha+1));`	Third part of function value
`val4=(((3/2)*(((block*4)*h)^(alpha/2)))-(((block*4)*h)^4))^3;`	Fourth part of functional value
`val5=(x(1,((block*4)+1)))^(3/2);`	Fifth part of functional value
`x(2,5)=val1-val2+val3+val4-val5;`	Final calculation of functional value of x(1,5) step
`start4=labBroadcast(4,x);`	Broadcast x to other labs
`end`	End of **if command** which began labindex==1
`else`	If not block 1 undertake these calculations
`if (block==1 && labindex>4)`	**If command** - all LABs that did not participate in starting phase
`start4=labBroadcast(4);`	Receive broadcast from LAB 4

x=start4;	Store result from 4 as new value for x matrix
end	end of **if command** which began (block==1 && labindex >4)
if ((block==1 && labindex>4) || (block>1))	**If command** - all LABs that did not participate in starting phase or if block is greater than 1
%Main phase,calculating the Lubich value	-
%xpost is value of the starting weight part of the equation	-
xpost=0;	Variable required for starting phase part of algorithm
%xcon is the value of the convolution part of equation	-
xcon=0;	Variable required for convolution part of algorithm
for m=1:5	**For loop** for starting phase elements already calculated in block 1 (or by other labs)
%Equation trying to solve.	-
res1=(convweights((n+2-m),1)*x(2,m));	Convolution weight multiplied by relevant functional value
xcon=xcon+res1;	Cumulative total for convolution weights mulitpled by functional value
if m<=(s+1)	**If command** for starting weight element of the algorithm
xpost=xpost+(startweight(1,m)*x(2,m));	Cumulative total for starting weight multiplied by relevant functional value
end	End of **if command** which began m<=(s+1)
end	End of **for loop** which began m=1:5
%predicting the value at current step prior to calculation	-
%through algorithm	-
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%	-
%Receiving values from other labs	-
if labindex==2	**If command** for LAB 2 only
if (block==2)	**If command** for block 2 only
Sendforward8=labBroadcast(8);	Receive broadcast from LAB 8
x=Sendforward8;	Store result from 8 as new value for x matrix
end	End of **if command** which began with block==2
Sendforward1=labBroadcast(1);	Receive broadcast from LAB 1 (x value and function)
x(:,(n-(labindex-2)))=Sendforward1;	Store result from LAB 1 in appropriate element of x matrix

`elseif labindex==3`	else for LAB 3 only
`if (block==2)`	**If command** for block 2 only
`Sendforward8=labBroadcast(8);`	Receive broadcast from LAB 8
`x=Sendforward8;`	Store result from 8 as new value for x matrix
`end`	End of **if command** which began with block==2
`Sendforward1=labBroadcast(1);`	Receive broadcast from LAB 1 (x value and function)
`x(:,(n-(labindex-2)))=Sendforward1;`	Store result from LAB 1 in appropriate element of x matrix
`x(:,(n-(labindex-3)))=labReceive(2,2);`	Receive and store data from LAB 2 (x value and function)
`elseif labindex==4`	else for LAB 4 only
`if (block==2)`	**If command** for block 2 only
`% if (block>1 && (block~=(ceil(N/8)+1)))`	-
`Sendforward8=labBroadcast(8);`	Receive broadcast from LAB 8
`x=Sendforward8;`	Store result from 8 as new value for x matrix
`end`	End of **if command** which began with block==2
`Sendforward1=labBroadcast(1);`	Receive broadcast from LAB 1 (x value and function)
`x(:,(n-(labindex-2)))=Sendforward1;`	Store result from LAB 1 in appropriate element of x matrix
`x(:,(n-(labindex-3)))=labReceive(2,2);`	Receive and store data from LAB 2 (x value and function)
`x(:,(n-(labindex-4)))=labReceive(3,3);`	Receive and store data from LAB 3 (x value and function)
`elseif labindex==5`	else for LAB 5 only
`if (block~=1)`	**If command** for blocks not equal to 1
`Sendforward1=labBroadcast(1);`	Receive broadcast from LAB 1 (x value and function)
`x(:,(n-(labindex-2)))=Sendforward1;`	Store result from LAB 1 in appropriate element of x matrix
`x(:,(n-(labindex-3)))=labReceive(2,2);`	Receive and store data from LAB 2 (x value and function)
`x(:,(n-(labindex-4)))=labReceive(3,3);`	Receive and store data from LAB 3 (x value and function)
`x(:,(n-(labindex-5)))=labReceive(4,4);`	Receive and store data from LAB 4 (x value and function)
`end`	End of **if command** which began block~=1
`elseif labindex==6`	else for LAB 6 only
`if (block~=1)`	**If command** for blocks not equal to 1
`Sendforward1=labBroadcast(1);`	Receive broadcast from LAB 1 (x value and function)

x(:,(n-(labindex-2)))=Sendforward1;	Store result from LAB 1 in appropriate element of x matrix
x(:,(n-(labindex-3)))=labReceive(2,2);	Receive and store data from LAB 2 (x value and function)
x(:,(n-(labindex-4)))=labReceive(3,3);	Receive and store data from LAB 3 (x value and function)
x(:,(n-(labindex-5)))=labReceive(4,4);	Receive and store data from LAB 4 (x value and function)
end	End of **if command** which began block ~=1
x(:,(n-(labindex-6)))=labReceive(5,5);	Receive and store data from LAB 5 (x value and function)
elseif labindex==7	else for LAB 7 only
if (block~=1)	**If command** for blocks not equal to 1
Sendforward1=labBroadcast(1);	Receive broadcast from LAB 1 (x value and function)
x(:,(n-(labindex-2)))=Sendforward1;	Store result from LAB 1 in appropriate element of x matrix
x(:,(n-(labindex-3)))=labReceive(2,2);	Receive and store data from LAB 2 (x value and function)
x(:,(n-(labindex-4)))=labReceive(3,3);	Receive and store data from LAB 3 (x value and function)
x(:,(n-(labindex-5)))=labReceive(4,4);	Receive and store data from LAB 4 (x value and function)
end	End of **if command** which began block ~=1
x(:,(n-(labindex-6)))=labReceive(5,5);	Receive and store data from LAB 5 (x value and function)
x(:,(n-(labindex-7)))=labReceive(6,6);	Receive and store data from LAB 6 (x value and function)
elseif labindex==8	else for LAB 8 only
if (block~=1)	**If command** for blocks not equal to 1
Sendforward1=labBroadcast(1);	Receive broadcast from LAB 1 (x value and function)
x(:,(n-(labindex-2)))=Sendforward1;	Store result from LAB 1 in appropriate element of x matrix
x(:,(n-(labindex-3)))=labReceive(2,2);	Receive and store data from LAB 2 (x value and function)
x(:,(n-(labindex-4)))=labReceive(3,3);	Receive and store data from LAB 3 (x value and function)
x(:,(n-(labindex-5)))=labReceive(4,4);	Receive and store data from LAB 4 (x value and function)
end	End of **if command** which began block ~=1
x(:,(n-(labindex-6)))=labReceive(5,5);	Receive and store data from LAB 5 (x value and function)
x(:,(n-(labindex-7)))=labReceive(6,6);	Receive and store data from LAB 6 (x value and function)
x(:,(n-(labindex-8)))=labReceive(7,7);	Receive and store data from LAB 7 (x value and function)
elseif (labindex==1 && block==2)	else if LAB =1 and in block 2

`Sendforward8=labBroadcast(8);`	Receive broadcast from LAB 8 (x value and function)
`x=Sendforward8;`	Store result from ALB 8 in appropriate element of x matrix
`end`	End of **if command** which began labindex==1
`xpre=0;`	Assignment of zero to xpre used in calculating predictor of current step
`for ipredict1=1:n`	**For loop** for cumulated predictor element
`xpre=xpre+(startcoef(1,(n+1-ipredict1))*x(2,ipredict1));`	Starting coefficient of predictor mulitplied by functional value of x at current iteration
`end`	End of **for loop** which began ipredict1=1:n
`%No value for the initial part of the predictor, this`	-
`%would change with different initial value x(1,1)`	-
`xpredict=(h^alpha)*xpre;`	Predicted value of x at current step
`val1=(40320/(gamma(9-alpha)))*((n*h)^(8-alpha));`	First part of functional value
`val2=3*((gamma(5+(alpha/2)))/(gamma(5-(alpha/2))))*((n*h)^(4-(alpha/2)));`	Second part of functional value
`val3=(9/4)*(gamma(alpha+1));`	Third part of function value
`val4=(((3/2)*((n*h)^(alpha/2)))-((n*h)^4))^3;`	Fourth part of functional value
`xpredictfunc=val1-val2+val3+val4-(xpredict^(3/2));`	Final calculation of predicted functional value
`if ((labindex>5 && block==1) || (block>1))`	**If command** for LABs greater than 5 and in block 1 OR any LAB who is not in block 1
`for otherlabval=6:n`	**For loop** from 6 to current step
`res2=(convweights((n+2-otherlabval),1)*x(2,otherlabval));`	Multiplication of convolution weight against current functional value of x
`xcon=xcon+res2;`	Cumulative total of above calculations
`end`	End of **for loop** which began otherlabval=6:n
`end`	End of **if command** which began ((labindex>5 && block==1)||(block>1))
`x(1,(n+1))=x(1,1)+((h^alpha)*(xcon+(convweights(1,1)*xpredictfunc)))+((h^alpha)*xpost);`	Calculated value of x(1,(n+1)) from previous calculations
`x(2,(n+1))=val1-val2+val3+val4-((x(1,(n+1)))^(3/2));`	Calculated value of function of x(1,(n+1))
`% disp('The value of x at this point is...')`	-
`% disp(labindex)`	-
`% disp(x(:,(n+1)))`	-
`%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%`	-

%Sending result to other labs and receiving broadcast from	-
%Lab 8	-
if labindex==1	**If command** for LAB 1 only
Sendforward1=labBroadcast(1,x(:,(n+1)));	Broadcast value and function of current x to other LABs
if (n+7)<=N	**If command** step + 7 is less than or equal to final step
Sendforward8=labBroadcast(8);	Receive broadcast from LAB 8
x=Sendforward8;	Replace current x matrix with one received from LAB 8
end	End of **if command** which began with (n+7)<=N
elseif labindex==2	else LAB 2 only
if (n+1)<=N	**If command** step + 1 is less than or equal to final step
labSend(x(:,(n+1)),3,2);	Send x(:,(n+1)) to LAB 3
if (n+2)<=N	**If command** step + 2 is less than or equal to final step
labSend(x(:,(n+1)),4,2);	Send x(:,(n+1)) to LAB 4
if (n+3)<=N	**If command** step + 3 is less than or equal to final step
labSend(x(:,(n+1)),5,2);	Send x(:,(n+1)) to LAB 5
if (n+4)<=N	**If command** step + 4 is less than or equal to final step
labSend(x(:,(n+1)),6,2);	Send x(:,(n+1)) to LAB 6
if (n+5)<=N	**If command** step + 5 is less than or equal to final step
labSend(x(:,(n+1)),7,2);	Send x(:,(n+1)) to LAB 7
end	End of **if command** which began with (n+5)<=N
end	End of **if command** which began with (n+4)<=N
end	End of **if command** which began with (n+3)<=N
end	End of **if command** which began with (n+2)<=N
end	End of **if command** which began with (n+1)<=N
labSend(x(:,(n+1)),8,2);	Send x(:,(n+1)) to LAB 8
if (n+6)<=N	**If command** step + 6 is less than or equal to final step
Sendforward8=labBroadcast(8);	Receive broadcast from LAB 8

x=Sendforward8;	Store and replace x with matrix received from LAB 8
end	End of **if command** which began with (n+6)<=N
elseif labindex==3	else LAB 3 ONLY
if (n+1)<=N	**If command** step + 1 is less than or equal to final step
labSend(x(:,(n+1)),4,3);	Send x(:,(n+1)) to LAB 4
if (n+2)<=N	**If command** step + 2 is less than or equal to final step
labSend(x(:,(n+1)),5,3);	Send x(:,(n+1)) to LAB 5
if (n+3)<=N	**If command** step + 3 is less than or equal to final step
labSend(x(:,(n+1)),6,3);	Send x(:,(n+1)) to LAB 6
if (n+4)<=N	**If command** step + 4 is less than or equal to final step
labSend(x(:,(n+1)),7,3);	Send x(:,(n+1)) to LAB 7
end	End of **if command** which began with (n+4)<=N
end	End of **if command** which began with (n+3)<=N
end	End of **if command** which began with (n+2)<=N
end	End of **if command** which began with (n+1)<=N
labSend(x(:,(n+1)),8,3);	Send x(:,(n+1)) to LAB 8
if (n+5)<=N	**If command** step + 5 is less than or equal to final step
Sendforward8=labBroadcast(8);	Receive broadcast from LAB 8
x=Sendforward8;	Store and replace x with matrix received from LAB 8
end	End of **if command** which began with (n+5)<=N
elseif labindex==4	else LAB 4 ONLY
if (n+1)<=N	**If command** step + 1 is less than or equal to final step
labSend(x(:,(n+1)),5,4);	Send x(:,(n+1)) to LAB 5
if (n+2)<=N	**If command** step + 2 is less than or equal to final step
labSend(x(:,(n+1)),6,4);	Send x(:,(n+1)) to LAB 6
if (n+3)<=N	**If command** step + 3 is less than or equal to final step

`labSend(x(:,(n+1)),7,4);`	Send x(:,(n+1)) to LAB 7
`end`	End of **if command** which began with (n+3)<=N
`end`	End of **if command** which began with (n+2)<=N
`end`	End of **if command** which began with (n+1)<=N
`labSend(x(:,(n+1)),8,4);`	Send x(:,(n+1)) to LAB 8
`if (n+4)<=N`	**If command** step + 4 is less than or equal to final step
`Sendforward8=labBroadcast(8);`	Receive broadcast from LAB 8
`x=Sendforward8;`	Store and replace x with matrix received from LAB 8
`end`	End of **if command** which began with (n+4)<=N
`elseif labindex==5`	else LAB 5 ONLY
`if (n+1)<=N`	**If command** step + 1 is less than or equal to final step
`labSend(x(:,(n+1)),6,5);`	Send x(:,(n+1)) to LAB 6
`if (n+2)<=N`	**If command** step + 2 is less than or equal to final step
`labSend(x(:,(n+1)),7,5);`	Send x(:,(n+1)) to LAB 7
`end`	End of **if command** which began with (n+2)<=N
`end`	End of **if command** which began with (n+1)<=N
`labSend(x(:,(n+1)),8,5);`	Send x(:,(n+1)) to LAB 8
`if (n+3)<=N`	**If command** step + 3 is less than or equal to final step
`Sendforward8=labBroadcast(8);`	Receive broadcast from LAB 8
`x=Sendforward8;`	Store and replace x with matrix received from LAB 8
`end`	End of **if command** which began with (n+3)<=N
`elseif labindex==6`	else LAB 6 ONLY
`if (n+1)<=N`	**If command** step + 1 is less than or equal to final step
`labSend(x(:,(n+1)),7,6);`	Send x(:,(n+1)) to LAB 7
`end`	End of **if command** which began with (n+1)<=N
`labSend(x(:,(n+1)),8,6);`	Send x(:,(n+1)) to LAB 8

`if (n+2)<=N`	**If command** step + 2 is less than or equal to final step
`Sendforward8=labBroadcast(8);`	Receive broadcast from LAB 8
`x=Sendforward8;`	Store and replace x with matrix received from LAB 8
`end`	End of **if command** which began with (n+2)<=N
`elseif labindex==7`	else LAB 7 ONLY
`labSend(x(:,(n+1)),8,7);`	Send x(:,(n+1)) to LAB 8
`if (n+1)<=N`	**If command** step + 1 is less than or equal to final step
`Sendforward8=labBroadcast(8);`	Receive broadcast from LAB 8
`x=Sendforward8;`	Store and replace x with matrix received from LAB 8
`end`	End of **if command** which began with (n+1)<=N
`elseif labindex==8`	else LAB 8 ONLY
`Sendforward8=labBroadcast(8,x);`	Broadcast x matrix to other LABs
`end`	End of **if command** which began labindex==1
`end`	End of **if command** which began ((block==1 && labindex>4) || (block>1))
`end`	End of **if command** which began (block==1 && labindex<5)
`else`	Alternative option for when n is not less than or equal to final step
`%Receiving data from Lab 1`	-
`Sendforward1=labBroadcast(1);`	Receive broadcast from LAB 1 (x value and function)
`x(:,(n-(labindex-2)))=Sendforward1;`	Store result from LAB 1 in appropriate element of x matrix
`if (block==2 && labindex<5)`	**If command** for small number of steps where some labs have ceased to process data beyond block 1
`Sendforward8=labBroadcast(8);`	Receiving broadcast from LAB 8
`x=Sendforward8;`	Store broadcast from 8 as x matrix
`end`	End of **if command** which began (block==2 && labindex<5)
`if labindex==8`	**If command** for LAB 8
`if (n-6)<=N`	**If command** when the current step - 6 is sless than or equal to final step.
`x(:,(n-(labindex-3)))=labReceive(2,2);`	Receive result for x and its function from LAB 2
`if (n-5)<=N`	**If command** when the current step - 5 is sless than or equal to final step.

`x(:,(n-(labindex-4)))=labReceive(3,3);`	Receive result for x and its function from LAB 3
`if (n-4)<=N`	**If command** when the current step - 4 is sless than or equal to final step.
`x(:,(n-(labindex-5)))=labReceive(4,4);`	Receive result for x and its function from LAB 4
`if (n-3)<=N`	**If command** when the current step - 3 is sless than or equal to final step.
`x(:,(n-(labindex-6)))=labReceive(5,5);`	Receive result for x and its function from LAB 5
`if (n-2)<=N`	**If command** when the current step - 2 is sless than or equal to final step.
`x(:,(n-(labindex-7)))=labReceive(6,6);`	Receive result for x and its function from LAB 6
`if (n-1)<=N`	**If command** when the current step - 1 is sless than or equal to final step.
`x(:,(n-(labindex-8)))=labReceive(7,7);`	Receive result for x and its function from LAB 7
`end`	end of **if command** which began (n-1)<=N
`end`	end of **if command** which began (n-2)<=N
`end`	end of **if command** which began (n-3)<=N
`end`	end of **if command** which began (n-4)<=N
`end`	end of **if command** which began (n-5)<=N
`end`	end of **if command** which began (n-6)<=N
`end`	end of **if command** which began labindex==8
`end`	End of **if command** which began with n<=N
`end`	End of **for loop** which began with block=1:(ceil(N/8))
`% if labindex==8`	(These results are commented out but would be reactivated in 'normal' circumstances to show result)
`% % disp('The value of x is...')`	-
`% disp(x(1,(N+1)))`	-
`% end`	-
`end`	End of **SPMD command**
`% delete(gcp('nocreate'))`	Alternative way to close the MatLab Pool, used in newer versions of MatLab Parallel Computing Toolbox
`matlabpool close`	Closing the MatLab Pool.
`toc;`	Stops the clock and displays time taken for program to execute.
`end`	End of **for loop** which began cloop=1:13 (used to run the program 13 times)

E. MATLAB PROGRAMS - CD

Attached to the back page of this thesis is a CD containing the MatLab Parallel Computing Toolbox programs used within this research.

BIBLIOGRAPHY

[1] J. Albanese, W. Sommemreich, *Network Security Illustrated*, McGraw-Hill, New York, 2004

[2] G.S. Almasi, A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings Publishing Company, Redwood City, 1994

[3] U.M. Ascher, *Numerical methods for evolutionary differential equations*, SIAM, Philadelphia, 2008

[4] D. Baleanu, K. Diethelm, E. Scalas, J.J. Trujillo, *Fractional Calculus: Models and Numerical Methods*, World Scientific, London, 2012

[5] N.E. Banks, *A Numerical Algorithm for Fractional Delay Differential Equations*, MSc Thesis, University of Chester (University of Liverpool), 2007

[6] B. Barney, *Introduction to Parallel Computing*, Available at: https://computing.llnl.gov/tutorials/parallel_comp/, 2014 (updated)

[7] L. Blank, *Numerical Treatement of Differential Equations of Fractional Order*, Numerical Analysis Report No. **287**, Manchester Centre for Computational Mathematics, 1996

[8] J.M. Brooke, M.S. Parkin, *Enabling scientific collaboration on the Grid*, Future Generation Computer Systems, **26**, 521-530, 2010

[9] L. Brugnano, D. Trigiante, *Solving Differential Problems by Multistep Initial and Boundary Value Methods*, Gordon and Breach Science Publishers, Amsterdam, 1998

[10] K. Burrage, *Parallel and Sequential Methods for Ordinary Differential Equations*, Clarendon Press, Oxford, 1995

[11] Cornell University Workshop, *Parallel Programming Concepts and High-Performance Computing*, Available at: https://www.cac.cornell.edu/VW/parallel/default.aspx, 2015

[12] H.M. Deitel, P.J. Deitel, *Java: How to program*, Prentice-Hall, Upper Saddle River, London, 1998

[13] K. Diethelm, *An algorithm for the numerical solution of differential equations of fractional order*, Electonic Transactions on Numerical Analysis, **5**, 1-6, 1997

[14] K. Diethelm, *An efficient parallel algorithm for the numerical solution of fractional differential equations*, Fractional Calculus & Applied Analysis, **14 No. 3**, 475-490, 2011

[15] K. Diethelm, J.M. Ford, N.J. Ford, M. Weilbeer, *Pitfalls in fast numerical solvers for fractional differential equations*, Elsevier Journal of Computational and Applied Mathematics, **186**, 482-503, 2006

[16] K. Diethelm, *The Analysis of Fractional Differential Equations: An application-Orientated Exposition Using Differential Operators of Caputo Type*, Springer, Berlin, 2010

[17] K. Diethelm, N.J. Ford, *Analysis of Fractional Differential Equations*, Journal of Mathematical Analysis and Applications, **265**, 229-248, 2002

[18] K. Diethelm, N.J. Ford, A.D. Freed, *Detailed error analysis for a fractional Adams method*, Numerical Algorithms, **36 Issue 1**, 31-52, 2004

[19] K. Diethelm, N.J. Ford, A.D. Freed, Y. Luchko, *Algorithms for the fractional calculus: A selection of numerical methods*, Computer Methods in Applied Mechanics and Engineering, **194**, 743-773, 2005

[20] K. Diethelm, A.D. Freed, *The FracPECE Subroutine for the Numerical Solution of Differential Equations of Fractional Order* in S. Heinzel, T. Plesser (*Eds.*) *Forschung und wissenschaftliches Rechnen 1998* **no. 53** in *GWDG-Berichte* (*Gesellschaft für wissenschaftliche Datenverarbeitung, Göttingen*), 57 - 71, 1999

[21] J. Dongarra, I. Foster I, G. Fox, W. Gropp, K. Kennedy, L. Torczon, A. White, *Sourcebook of Parallel Computing*, Elsevier Science, San Francisco, 2003

[22] J.E. Dorband, *Introduction to Parallel Computing* (*presentation*), http://ct.gsfc.nasa.gov/summer2004/Dorband.pdf, July 2004

[23] A. Dubrau, L. Hendren, *Taming MatLab*, ACM SIGPLAN Notices - OOPSLA '12, **47 Issue 10**, 503 - 522, 2012

[24] I. Faragó, Á. Havasi, Z. Zlatev, *Efficient implementation of stable Richardson Extrapolation algorithms*, Elsevier Computers and Mathematics with Applications, **60**, 2309-2325, 2010

[25] N.J. Ford, A.C. Simpson, *The numerical solution of fractional differential equations: Speed versus accuracy*, Numerical Algorithms, **26**, 333-346, 2001

[26] I. Foster, *The Grid: A new infrastructure for 21st century science*, Physics Today, February 2002

[27] G.H. Golub, C.F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 2013

[28] W.R. Graham, *A Research and Development Strategy for High Performance Computing*, Executive Office of the President, Office of Science and Technology Policy, 1987

[29] The Grid Cafe, Available at: http://www.gridcafe.org/EN/, 2015

[30] E. Hairer, S.P. Nørsett, G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, Springer, New York, 2000

[31] Helping Engineers Learn Mathematics (HELM), *Workbook 32: Numerical Initial Value Problems*, Available at: http://www.qub.ac.uk/helm/HELM_2008/pages/workbooks_1_50_jan2008/Workbook32/32_2_lin_mltstp_methds.pdf

[32] J.L. Hennessy, D.A. Patterson, *Computer Organization and Design: the Hardware/Software Interface*, Morgan Kaufmann, Waltham, 2012

[33] B.R. Hunt, R.L. Lipsman, J.M. Rosenberg, (with Coombes K.R., Osborn J.E., Stuck G.J.), *A Guide to MATLAB for Beginners and Experienced Users*, Cambridge University Press, New York, 2001

[34] Indianna University, *What is the difference between a processor, a chip, a socket, and a core?*, Available at: https://kb.iu.edu/d/avfb

[35] Intel, *Introduction to Hyper-Threading Technology*, Intel Developer Zone, Available at: https://software.intel.com/en-us/articles/introduction-to-hyper-threading-technology, 2010

[36] Intel, *Frequently Asked Questions: Intel Multi-Core Processor Architecture*, Intel Developer Zone, Available at: https://software.intel.com/en-us/articles/frequently-asked-questions-intel-multi-core-processor-architecture#_The_move_to_dual/multi-core explain

[37] Intel, *Intel Xeon Processor E5345 (8M Cache, 2.33 GHz, 1333 MHz FSB)*, Intel Ark, Available at: http://ark.intel.com/products/28032, accessed 2015

[38] Intel, *Intel Xeon Processor W5580 (8M Cache, 3.20 GHz, 6.40 GT/s Intel QPI)*, Intel Ark, Available at: http://ark.intel.com/products/37113, access 2015

[39] D. Kinghorn, *Hyper-Threading may be Killing your Parallel Performance*, Available from Puget Systems at: https://www.pugetsystems.com/labs/articles/Hyper-Threading-may-be-Killing-your-Parallel-Performance-578/, 2014

[40] J.D. Lambert, *Numerical methods for Ordinary Differential Systems: The initial value problem*, Wiley, Chichester, 2000

[41] Ch. Lubich, *Fractional Linear Multistep Methods for Abel-Volterra Integral Equations of the Second Kind*, Mathematics of Computation, **45 No. 172**, 463-469, 1985

[42] Ch. Lubich, *Discretized Fractional Calculus*, SIAM Journal of Mathematical Analysis, **17 No. 3**, 704-719, 1986

[43] Ch. Lubich, *Convolution Quadrature Revisited*, BIT Numerical Mathematics, **44**, 503-514, 2004

[44] The MathWorks Inc, *MatLab Parallel Computing Toolbox 5 User's Guide*, Natick MA, 2011 (online)

[45] The MathWorks Inc, *MatLab Parallel Computing Toolbox: Users Guide R2014b*, Matick MA, 2014 (online)

[46] The MathWorks, *why are MATLAB functions slower the first time they are called?*, Available at: http://www.mathworks.com/matlabcentral/newsreader/view_thread/311994, 2011

[47] The MathWorks, *Run MATLAB on multicore and multiprocessor machines*, Available at: http://uk.mathworks.com/discovery/matlab-multicore.html, 2015

[48] The MathWorks, *Which MATLAB functions benefit from multithreaded computation?*, Available at: http://www.mathworks.com/matlabcentral/answers/95958-which-matlab-functions-benefit-from-multithreaded-computation, 2013

[49] S. Maxwell, *UNIX System Administration: A Beginner's Guide*, McGraw-Hill/Osborne, New York, 2002

[50] C. Moler, *Parallel MATLAB: Multiple Processors and Multiple Cores*, The MathWorks News and Notes, June 2007, Available at: http://uk.mathworks.com/company/newsletters/articles/parallel-matlab-multiple-processors-and-multiple-cores.html

[51] MPI Forum, *MPI-2: Extensions to the Message-Passing Interface*, Available at: http://www.mpi-forum.org/docs/mpi-2.0/mpi-20-html/mpi2-report.html, 2001

[52] K.B. Oldham, J. Spanier, *The Fractional Calculus: Theory and Applications of Differentiation and Integration to Arbitrary Order*, Dover Publications, New York, 2006

[53] I. Podlubny, *Fractional Differential Equations*, Academic Press, California, 1999

[54] I. Podlubny, *Mittag-Leffler Function*, 2005 (updated 2012), Available at: http://uk.mathworks.com/matlabcentral/fileexchange/8738-mittag-leffler-function?s_tid=srchtitle

[55] A. Ralston, P. Rabinowitz, *A First Course in Numerical Analysis*, McGraw-Hill, New York, 1978

[56] University of California, *seti@home*, Availabale at: http://setiathome.berkeley.edu/index.php, 2015

[57] L.F. Shampine, *Numerical solution of ordinary differential equations*, Chapman & Hall, New York, 1994

[58] G. Sharma, J. Martin, *MATLAB: A Language for Parallel Computing*, International Journal of Parallel Programming, **37**, 3-36, 2009

[59] R.W. Shonkwiler, L. Lefton, *An Introduction to Parallel and Vector Scientific Computing*, Cambridge University Press, Cambridge, 2006

[60] A.C. Simpson, *Numerical Methods for the Solution of Fractional Differential Equations*, PhD Thesis, University College Chester (University of Liverpool), 2001

[61] T.L. Sterling, *Beowulf Cluster Computing with Linux*, Scientific and Engineering Computation Series, Cambridge, 2002

[62] G.W. Stewart, *Matrix Algorithms Volume 1: Basic Decompositions*, SIAM, Philadelphia, 1998

[63] E. Strohmaier, J.J. Dongarra, H.W. Meuer, H.D. Simon, *Recent Trends in the Marketplace of High Performance computing*, CT Watch Quarterly, 2005

[64] Top500.org, *Top500 Lists November 2014*, Available at: http://www.top500.org/lists/2014/11/, 2015

[65] M. Weilbeer, *Efficient Numerical Methods for Fractional Differential Equations and their Analytical Background*, PhD Thesis, Braunschweig Technical University, 2005, published by Papierflieger, Clausthal-Zellerfeld, 2006